A PROGRAM LOGIC AND ITS APPLICATION IN FULLY VERIFIED SOFTWARE FAULT ISOLATION

by

Lu Zhao

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

August 2012

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Lu Zhao**

has been approved by the following supervisory committee members:

John Regehr	, Chair	**05/11/2012** Date Approved
Ganesh Gopalakrishnan	, Member	**05/11/2012** Date Approved
Matthew Flatt	, Member	**05/11/2012** Date Approved
Matthew Might	, Member	**05/11/2012** Date Approved
Magnus Myreen	, Member	**05/11/2012** Date Approved

and by **Al Davis**, Chair of the Department of **School of Computing**

and by Charles A. Wight, Dean of The Graduate School.

ABSTRACT

Trusted computing base (TCB) of a computer system comprises components that must be trusted in order to support its security policy. Research communities have identified the well-known minimal TCB principle, namely, the TCB of a system should be as small as possible, so that it can be thoroughly examined and verified. This dissertation is an experiment showing how small the TCB for an isolation service is based on software fault isolation (SFI) for small multitasking embedded systems.

The TCB achieved by this dissertation includes just the formal definitions of isolation properties, instruction semantics, program logic, and a proof assistant, besides hardware. There is not a compiler, an assembler, a verifier, a rewriter, or an operating system in the TCB. To the best of my knowledge, this is the smallest TCB that has ever been shown for guaranteeing nontrivial properties of real binary programs on real hardware.

This is accomplished by combining SFI techniques and high-confidence formal verification. An SFI implementation inserts dynamic checks before dangerous operations, and these checks provide necessary invariants needed by the formal verification to prove theorems about the isolation properties of ARM binary programs. The high-confidence assurance of the formal verification comes from two facts. First, the verification is based on an existing realistic semantics of the ARM ISA that is independently developed by Cambridge researchers. Second, the verification is conducted in a higher-order proof assistant—the HOL theorem prover, which mechanically checks every verification step by rigorous logic.

In addition, the entire verification process, including both specification generation and verification, is automatic. To support proof automation, a novel program logic has been designed, and an automatic reasoning framework for verifying shallow safety properties has been developed. The program logic integrates Hoare-style reasoning

and Floyd's inductive assertion reasoning together in a small set of definitions, which overcomes shortcomings of Hoare logic and facilitates proof automation. All inference rules of the logic are proven based on the instruction semantics and the logic definitions. The framework leverages abstract interpretation to automatically find function specifications required by the program logic. The results of the abstract interpretation are used to construct the function specifications automatically, and the specifications are proven without human interaction by utilizing intermediate theorems generated during the abstract interpretation. All these work in concert to create the very small TCB.

For Arlene, parents, and friends

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I want to thank my advisor Professor John Regehr for his support of my research, for his guidance on choosing topics and conducting research, and for his help in presenting my work. He has been a role model to me in all aspects of research.

I would like to thank my committee members, Ganesh Gopalakrishnan, Matthew Flatt, Matthew Might and Magnus Myreen for their comments and discussions of my work. I appreciate that Magnus has helped solving technical issues in the HOL theorem prover.

I am grateful for many inspiring discussions of theories and techniques with Guodong Li. His demonstration of using HOL to solve verification problems provided me with the initial momentum to pursue this research.

I am indebted to many others for comments, discussion and criticism: Yang Chen, Jianjun Duan and Jon Rafkind.

CHAPTER 1

INTRODUCTION

The trusted computing base (TCB) of a security-critical computer system comprises components of the system that must be trusted for supporting its security policy. It includes firmware, hardware, and software critical to protection and must be designed and implemented such that system elements excluded from it need not be trusted to maintain protection [23]. Research communities have identified the principle of a *minimal trusted computing base*: the TCB of a system should be as small as possible, because a small TCB is often simple enough to be analyzed and verified thoroughly, so that a high degree of assurance can be achieved in its ability to correctly enforce the security policy [23, 92, 94]. However, the TCB of a practical computer system is not minimal. It usually includes the operating system and compiler which are complex software systems and are not susceptible to thorough examination. This dissertation attempts to answer a research question: what is the minimal TCB that can provide isolation in embedded systems? In particular I focus on the software components of a TCB, assuming that the underlying hardware and firmware are trustworthy.

Isolation—the guarantee that one computation on a machine cannot affect other computations—is a fundamental system service supporting multiprogramming. Reliable isolation enables many useful kinds of coexistence; for example, users can safely run code downloaded from the Internet, servers belonging to mutually-distrusting parties can be run in different virtual machines on the same physical box, and embedded systems can be made smaller and cheaper by running critical and noncritical code on the same processor.

Isolation can be implemented in many ways, including using physical partitioning across processors, hardware-assisted address space management, type-safety at the

programming language level, capability-based systems, and software fault isolation (SFI). Each method has a different TCB. The TCB of memory management unit (MMU) based methods includes an operating system and runtime libraries. General-purpose operating systems have long used this approach to provide isolation among processes. Language-based methods such as type safety of the Java language add the language runtime and standard library into the TCB. These TCBs are quite large and very difficult to verify.

SFI enforces isolation by rewriting a binary program to insert checking code in front of every dangerous operation [105]. An operation is *dangerous* if it may violate a security policy, e.g., by writing to out-of-bounds storage or attempting to execute unauthorized code. The inserted code checks the legitimacy of a dangerous operation at runtime. If the operation is deemed as safe according to the policy, then the checking code allows execution to continue without doing anything else; otherwise, the checking code aborts the execution, leaving other computations unaffected. SFI traditionally uses a verifier to check the presence of required logic in rewritten binary code before the code is executed. In this implementation, the TCB of SFI includes the verifier and a compiler; the latter generates the binary code of the verifier from its source code.

In the context of critical embedded systems, either component may be too large and unreliable. For example, Google's Native Client uses an SFI implementation to isolate untrusted binary code downloaded from the Internet in a web browser environment [111]. However, a routine code refactoring converted the mechanism of enforcing control flow safety to a `nop` on the x86 platform [78]. In addition, compiler bugs are not uncommon, and this may result in unexpected results in the binary code of programs [110].

This dissertation argues that placing the verifier and compiler in the TCB is *not* necessary for isolation service provided through SFI. A very small TCB whose components are formally defined and verified enables systems such as the one depicted in Figure 1.1 to be trusted. The key property is that larger, noncritical components (GUIs, network stacks, etc.) can be provably isolated from smaller critical components

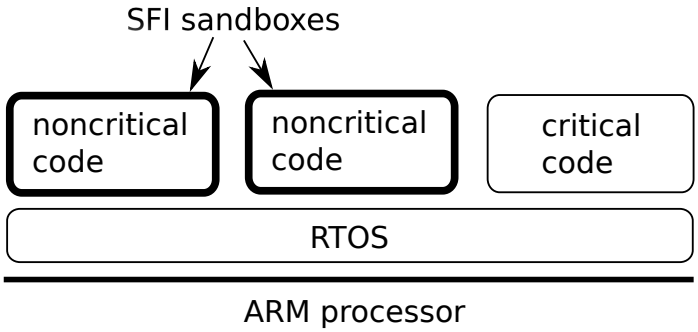


Figure 1.1: Multitasking embedded systems with provable isolation

(control loops, the RTOS, etc.) using SFI.

1.1 Thesis Statement

It is feasible to verify automatically in a mechanized proof assistant that a rewritten binary program respects its isolation policy with a minimal TCB, which includes just the formal definitions of isolation properties, instruction semantics, program logic, and the proof assistant.

This dissertation conducts research to support this thesis. The rest of this chapter will give a high-level overview of related fields and highlights the contents and contributions of the dissertation.

1.2 Binary Rewriting and SFI

Binary rewriting is a technique that takes a binary program generated by a compiler, modifies it according to certain requirements, and produces another binary program. The process may or may not be assisted with relocation information from the compiler, and it can be implemented either statically or dynamically. The newly generated program has enhancements which the original program does not have. For example, the technique can be used to perform whole-program optimization at link-time [16, 72, 101, 104]; it can also be used to enhance security characteristics of a program, e.g., reducing certain security vulnerabilities [9, 56, 87] and safeguarding a binary program from reverse engineering [62]; binary instrumentation uses it to detect, monitor and measure certain behaviors of a program [58]; binary translation applies it to generate a program that runs on a different architecture or operating

system [19, 99]. In addition, researchers have found its applications in other areas such as software caching [49], software transactional memory [84] and so on.

A particular application of binary rewriting is sandboxing: untrusted binary code may execute in a confined environment inside a host software system, such that the untrusted code cannot affect the host system except as specified by a well-defined policy. Sandboxing implemented statically is traditionally called software fault isolation (SFI) [28,66,105]. Sandboxing can also be implemented dynamically: a reference monitor intercepts and translates dangerous instructions at runtime [34,38]; in this case, there is no static instrumentation of the untrusted code.

This dissertation focuses on static binary rewriting techniques to implement sandboxing in embedded systems. There are two fundamental research issues in SFI. The first one is designing efficient isolation-enforcing mechanisms satisfying a security policy such that the runtime overhead of inserted code is minimal. Existing research has achieved impressive results. For example, it was reported that an average overhead less than 5% could be achieved for certain types of sandboxing [105, 111]. The second issue is guaranteeing the security correctness of the mechanisms and their implementation. This has not been adequately addressed. For instance, early attempts [29, 100] to design SFI mechanisms on the x86 architecture were found to have flaws more than a half decade later [66]. The security correctness of SFI is still argued by code review and testing, hoping to find bugs in the source code of its verifier [111]. Research efforts have been made to verify an SFI design formally [64, 108]. Although these efforts are big improvements compared to informal arguments, they are not satisfactory for important reasons. First, what they verified is designs at the conceptual level. They assumed the correctness of a verifier or a monitor. This means that the implementation of the most important components in a TCB is left out completely. Second, they modeled the semantics of a small subset of an instruction set architecture (ISA). Although it is informally arguable that a small subset may be representative to the actual execution of a real binary program, the history of formal verification has examples showing that using a sound and realistic semantics may avoid unnecessary errors. I emphasize the importance of semantics in Section 1.4.1.

This dissertation addresses the second issue of SFI by formally verifying the binary code that runs on a physical processor.

1.3 Isolation Policy

As a security mechanism, SFI needs a security policy as a reference for its implementation. This dissertation includes two safety properties in the security policy: *memory safety*: store operations are confined to predefined regions, and *control flow integrity*: execution may not escape a predetermined control flow graph (CFG). These are sufficient for isolation purposes, i.e., a computation module with these two properties cannot affect other computations in the same address space. However, the control flow integrity property is not a necessary condition for sandboxing, because there exist more relaxed policies. For example, several existing SFI mechanisms enforce a segment-based policy for control flow transfers: all jumps of a program are limited into a continuous address segment allocated for the code of the program. Recent studies show that such a loose policy alone allows certain types of attacks to happen such as return-oriented programming [14, 98]. Therefore, this dissertation adopts the strict policy for control flow transfers, and the control flow integrity stated above disables those attacks.

The policy not only decides security features of a mechanism but also affects the performance overhead of inserted code. For example, the loose policy mentioned above can restrict an indirect jump with one or two additional instructions using a masking operation, but the simple masking operation cannot enforce control flow integrity. More complicated techniques must be used, and they result in more overhead.

Each property in the policy accepts a parameter that is defined in advance. The memory safety property requires a set of memory regions that are allocated ahead of time for a program to modify. These regions are normally specified by designers and developers of embedded systems. The control flow integrity property needs a reference CFG of the program. The CFG dictates the control flow transfers that are allowed for every instruction of the program. It may be written down by hand or created by a binary analysis tool.

The security policy is external to the system that enforces and verifies the policy. It is related to the usefulness of the system and the success of the verification. For instance, if a policy is not correctly given with respect to the execution of a program, e.g., the given CFG is too small, then the policy enforcing mechanism will prevent many expected control flow transfers from taking place, resulting in a nonfunctioning program. The corresponding verification will fail, too, because the program must violate the CFG policy in order to execute some code branches. In contrast, if the given CFG is larger than necessary, then a program may be able to execute more code than desirable under this loose policy, and the verification process confirms that the program does not violate the policy. Similar discussions also apply for the memory safety policy. Ideally, a policy should match the expected behaviors of a program, granting the least privilege to the program in order for it to accomplish its tasks—the so called principle of least privilege [92, 94].

This dissertation does not discuss how the security policy is created, namely, how the two parameters of the isolation properties are given. It makes basic assumptions about the content of the policy and focuses on enforcing mechanisms and a formal system that verifies the compliance of a rewritten program to the policy.

1.4 Formal Verification

Formal verification, in its general meaning, refers to using formal mathematical models to describe software specification and implementation as well as to verify that the implementation meets requirements of the specification. Many methods that apply mathematical models may be included in the broad area of formal verification, such as abstract interpretation, model checking, program modeling and synthesizing, theorem proving etc.. This dissertation focuses on higher-order logic theorem proving: using a mechanized proof assistant, a.k.a. a theorem prover, to verify safety properties of binary programs automatically.

1.4.1 Theorem Proving

Since the invention of the LCF system by Robin Milner, using a mechanized proof assistant has become the norm in theorem proving [39,69]. The basic approach

of theorem proving is formalizing the syntax and semantics of program constructs as logic formulas, whose basic elements are accepted as definitions in the proof assistant. These definitions are used to derive theorems for inference rules in the logic system supported by the assistant, and facts about a program are deduced as theorems by using the inference rules [40, 41].

The advantages of this approach have been widely apprehended in research communities. First, the reasoning process for inference rules is based on the semantics of program constructs. This guarantees the correctness of inference rules. This is very important, because rules defined directly might not be correct. For example, the original assignment rule used by Hoare was not correct in terms of program semantics [40]; one of the goals of the foundational proof-carrying code project was to prove, based on semantics, the typing rules for typed assembly language, which were previously defined directly in proof-carrying code [5, 6, 71, 102]. Second, the mechanized proof assistant guarantees that a theorem can only be deduced from existing theorems by using sound inference rules. This reduces the amount of trusted code to basic definitions. Once the definitions are correct, the correctness of program properties is machine-checked and guaranteed by the proof assistant. This is in contrast with other non-theorem-proving based verification methods, such as abstract interpretation or program synthesis [10, 86, 106]. In these methods, a proof is implicitly embedded in the implementation of the corresponding analysis or synthesis tools [21]. These tools are complex pieces of software, and there are no better methods known to verify their correctness other than using a mechanized theorem prover [89]. Because of the rigorous logic foundation and machine-checked proof, using a mechanized theorem prover has been regarded as the most trustworthy approach in formal verification [48, 55, 89].

One of the major disadvantages of using a mechanized higher-order logic proof assistant is its manual proving process: it depends on human intelligence to apply correct tactics, because there are no automatic decision procedures for higher-order logic formulas. The current proof automation in such systems relies on fixed proof steps, which might be programmed according to certain patterns [77].

1.4.2 ARM Semantics

Formal semantics plays a very important role in theorem proving. In the context of binary programs, it interprets what individual machine instructions do, and this echoes the semantics of a high-level language described in the traditional program logic literature [25, 41]. The correctness of every inference rule and specification is stripped down to and interpreted by the semantics. However, developing a sound and realistic formal semantics is not trivial, and research efforts have been made to build such semantics for common ISAs such as x86 and ARM [35, 37, 70, 85]. This dissertation utilizes the existing ARM formal semantics that has been independently developed by Cambridge researchers [37].

Originally, this ARM semantics is modeled as an operational semantics in the HOL theorem prover. Fox verified that the semantics was correctly implemented by a particular ARM chip [35]. Based on the operational semantics, Myreen developed an axiomatic semantics for the ARM ISA [75]. The axiomatic semantics is formally proven from the operational semantics in the HOL system and has a more concise representation of machine states than does the original operational semantics. This dissertation uses this axiomatic semantics in its formal verification.

1.4.3 The HOL Proof Assistant

The HOL system is a mechanized higher-order logic theorem prover [39, 42]. The core of its logic system is an implementation of typed λ-calculus [18, 47]. Throughout decades of development, many useful theories have been formalized and proven in the system; e.g., the word or bit-vector theory can faithfully model the bounded integer semantics of registers and memory. Several useful definition tools have been integrated into the system, such as the inductive relation definition. The system supports both forward rule-based proof and backward goal-directed proof. Proof steps are carried out by applying tactics, rules, or custom-built SML programs. I used the HOL proof assistant to conduct the formal verification work presented in this dissertation.

Inductive relation definition is very useful in my dissertation research. It defines a new relation according to given patterns, and the new relation may be recursive [68].

A very desirable property of this definition is that instances of the newly defined relation can only be constructed according to the definition patterns. This technique is used in several places. For example, a single rule may be defined in Section 4.4 to convert an existing relation to another that uses different logic constants while keeping the same underlying interpretation; the function judgment is defined recursively with a Base rule and an Induction rule in Section 4.6.1.2.

1.5 ARMor Toolchain

This dissertation supports the thesis stated in Section 1.1 by presenting the theory and practice of ARMor: the first toolchain that implements SFI and formally verifies the isolation policies for a rewritten ARM binary program.

I designed and implemented ARMor's SFI-enforcing mechanisms by using static binary rewriting techniques based on the Diablo framework. Diablo is a link-time optimizer or rewriter, and it can analyze and transform statically linked executables to achieve better performance, smaller code size and improved security [88, 104]. I utilized its API to develop a set of binary transformations to enforce the isolation properties—the memory safety and the control flow integrity discussed in Section 1.3—on ARM executables emitted by GCC.

I verified the isolation properties of the rewritten program in the HOL proof assistant automatically, based on the existing formal semantics of the ARM ISA introduced in Section 1.4.2. The final result of the verification is a theorem stating that the rewritten program does not violate the memory safety and the control flow integrity.

1.5.1 ARMor's Isolation Policy

As discussed in Section 1.3, creating safety policies is not the topic of this dissertation, but some reference policies are needed in order to implement and verify a security mechanism. For the purposes of this dissertation, I specify fixed memory regions as the reference policy for the memory safety property and use the CFG that Diablo computes as the reference CFG for the control flow integrity property.

Verifying that a program respects a CFG policy computed by some binary rewrit-

ing tool is useful for security-critical programs, because such tool makes assumptions about the execution of the program when it computes a CFG, and these assumptions can be easily violated by subtle issues in source languages or in compilers. For example, buffer overflow can cause a return address saved on the stack to be overwritten, resulting in a control flow transfer to unauthorized code when a function returns [3,15]. This type of unexpected control flow transfer is not in the legal CFG computed by a tool, but it can happen and be utilized by attackers. ARMor's verification exposes a huge amount of low-level details of a binary program to scrutiny and guarantees that given a reference CFG policy, the program does not violate it, thus preventing any kind of control-hijacking attack.

1.5.2 High-level Assurance

ARMor provides a high-confidence argument about isolation by defining the isolation properties formally and verifying mechanically that the properties hold in the verified binary program. ARMor's verification is not conducted at a high level, such as at an abstraction, source code, or even assembly level; instead, it is carried out at the lowest level of an implementation—the binary code that runs on a physical processor. This level of formal verification leaves an extremely small TCB, which is subject to thorough examination.

1.5.3 Automated Proof

ARMor completes the entire verification process automatically: it not only verifies specifications of a program automatically, but also generates the specifications automatically. It achieves this level of automation for proving shallow safety properties by using a carefully designed logic and by using abstract interpretation.

It is noteworthy that in formal verification, proof automation is considered as a process that verifies a specification against an implementation based on semantics. It does not include the generation of the specification, because a common idiom is that the specification comes from user requirements, and it is not part of the proof automation. For example, existing research in reasoning about low-level programs verifies specifications generated manually [83, 113]. However, I take a different stand

in reasoning about machine-code programs, because it is extremely difficult, or impractical, to write down correct specifications for binary programs manually.

1.5.4 Processing Steps

At a very high level, ARMor operates as follows:

1. An ARM executable is created by a compiler.
2. An extension for Diablo, which I developed, sandboxes the executable.
3. The HOL proof assistant automatically verifies that the rewritten executable conforms to the memory safety and control flow integrity policies. This step does not require human interaction.

Figure 1.2 depicts the processing steps of ARMor in more detail. The SFI implementation provides a rewritten binary program to the formal verification framework; as mentioned in Section 1.5.1, it also provides the control flow information as the reference CFG policy.

To describe the verification process roughly, Hoare-style reasoning is used to produce judgments about code blocks, whereas the safety properties of a program are defined as part of the instruction semantics and thus are guaranteed at every point in the program. Next, function invariants in the form of derivation relations among the Hoare judgments of code blocks are discovered using abstract interpretation. This process is recursively performed for all functions in the program, starting from the leaf functions up to the entry function in the call graph. Finally, function judgments

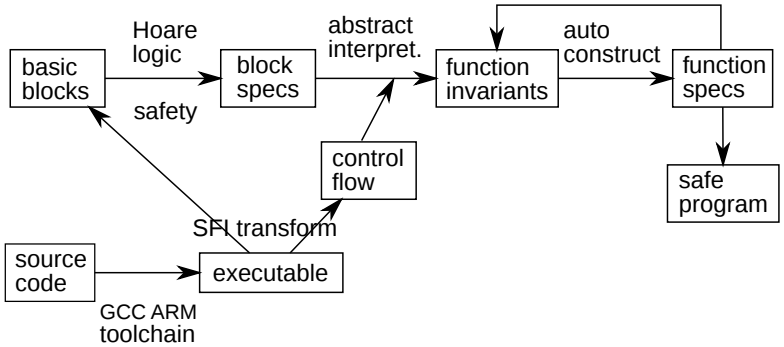


Figure 1.2: Processing steps of ARMor

are built based on the result of the abstract interpretation and then proven correct. The judgment of the entry or top-level function is the final program judgment, which ensures the safety properties in the verified program. The entire process will be described in detail in Chapter 6.

1.5.5 Bug Finding

ARMor is not designed to find bugs, and its purpose is providing the high-level assurance that the isolation service provided in Figure 1.1 is bug-free. A bug in application code or in the binary rewriter or in the compiler can result in verification failure, or is guaranteed by ARMor's verification that it cannot affect the rest of the system—an ARMor-level trap that stops the faulting computation. Ideally, these bugs are discovered during predeployment testing.

1.6 Contributions

This dissertation makes the following contributions.

1. It demonstrates that the TCB of an isolation service based on SFI may be extremely small, including just the formal definitions of isolation properties, instruction semantics, program logic, and a theorem prover, besides underlying hardware. In particular, the TCB does not include an operating system, a compiler, a rewriter, or a verifier. To the best of my knowledge, this is the smallest TCB that has ever been shown for guaranteeing nontrivial properties of realistic binary programs.

2. It implements an SFI strategy for small embedded systems which provides provably reliable isolation for a computation task.

3. It designs a novel and practical program logic $\mathcal{L}_{fn}$. This logic integrates the ideas of rule-based Hoare style reasoning and of Floyd's inductive assertion-style reasoning in a single set of logic definitions. The result is that the logic does not require complicated rule systems, that it unifies the treatment of partial and total correctness specifications, and that it supports function-level modular reasoning.

4. Based on $\mathcal{L}_{fn}$, it develops an automated framework for proving safety properties of binary programs in a higher-order logic proof assistant. This framework uses abstract interpretation with logic domains to discover the program specifications automatically. A verification engine builds logic terms and completes a proof by taking the results of the abstract interpretation and instantiating logic parameters in program judgments. This framework is applied to automatically verify the isolation properties of rewritten binary programs in the HOL system.

Although I take the ARM semantics as an example of utilizing existing formal semantics in my research implementation, most of the formal verification framework is architecture-neutral, because the semantics is parametrized in $\mathcal{L}_{fn}$, and it is straightforward to instantiate the semantic parameter with another ISA's axiomatic semantics.

1.7 A Motivating Example

Due to the abstract nature of topics, I use a concrete example shown in Figure 1.3(a) to illustrate the logic concepts and formalizations presented in this dissertation. The program has two functions: `entryFun` and `foo`; `entryFun` is the entry or top-level function, and its first block calls `foo`. The code of each instruction is paired with the address of the instruction to get a direct association with code assertions in ARMor's logic. Suppose that we verify its isolation properties as described in Section 1.3. The security policy is given in Figure 1.3(b). Denote all writable memory regions as a set of addresses `mem`, and use a function, `succ`, to model the reference CFG, which returns the set of addresses where the control may go when given an instruction address. The last lambda expression in the reference CFG, $\lambda a.\{a + 4\}$, represents the control flow transfers within a basic block in which the value of PC is increased by 4 at each instruction—the length of instructions of the ARM ISA is fixed to 4 bytes in the ARM mode. Figure 1.4(a) shows the CFG policy at the basic block level.

ARMor verifies this program against its isolation policy automatically and renders the following proven theorem in the HOL proof assistant as its final verification result:

```
<entryFun>
blk1: (0x0, 0xE3A0D441) //mov R13,#0x41000000
      (0x4, 0xE3A00000) //mov R0,#0
      (0x8, 0xE1A01000) //mov R1,R0
      (0xC, 0xEB000000) //bl foo (branch to foo)
blk2: (0x10,0xEAFFFFFE) //b +#0 (branch to blk2)
<foo>
blk3: (0x14,0xE2411001) //sub R1,R1,#0x1
      (0x18,0xE3320101) //teq R2,#0x40000000
      (0x1C,0x11A0F00E) //movne PC,R14 (return not equal)
blk4: (0x20,0xE5C21000) //strb R1,[R2] (store byte)
      (0x24,0xE3310000) //teq R1,#0x0 (test equal)
      (0x28,0x1AFFFFF9) //bne foo (branch not equal)
blk5: (0x2C,0xe1a0f00e) //mov PC,R14 (return)
```

(a) Program code

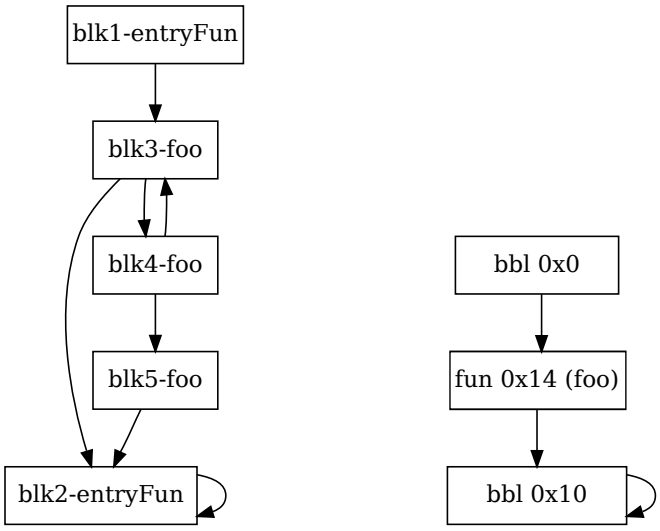
Memory safety policy `mem`	
$\{a \mid 0x40000000 \leq a \wedge a < 0x40001000\}$	
CFG policy `succ`	
input	output
0xC	{0x14}
0x10	{0x10}
0x1C	{0x10, 0x20}
0x28	{0x14, 0x2C}
0x2C	{0x10}
others	$\lambda a.\{a + 4\}$

(b) Safety policies

Figure 1.3: An illustrating example

```
PROG_SPEC SAFE_INS entryFunction 0x0 pred bspec          (1.1)
```

where `entryFunction` is a set of nodes of the `entryFun` function, which includes two nodes for basic blocks—blk1 and blk2—and one node for the abstraction of the `foo` function. Figure 1.4(b) depicts these nodes and the control flow transfer relation among them. The term `0x0` is the entry address of the top-level function. The term `pred` models the given CFG at the node level in terms of the predecessor relation: it takes a node and returns its predecessor nodes. The last term `bspec` is the specification for the top-level function, which maps each node to the precondition of



(a) Block level CFG (b) `entryFun` nodes

Figure 1.4: Program CFG

the node. The `SAFE_INS` relation encodes the isolation properties at every instruction of the program. In plain English, Theorem (1.1) states that for every instruction of the program in Figure 1.3(a), it respects, or does not violate, the memory safety and control flow integrity policies given in Figure 1.3(b). This dissertation will explain how to reach the theorem from the given program with respect to its isolation policy.

In the verification framework, the only architecture-dependent element is the `SAFE_INS` relation, which formally defines the safety properties of interest in semantics—the isolation properties in this example. For different safety properties, the same framework can be reused by defining a new relation. All other logic definitions and proven rules stay unchanged.

CHAPTER 2

RELATED WORK

This chapter discusses related work in SFI and theorem proving. Additional discussion will follow in later chapters.

2.1 SFI

There have been many SFI designs and implementations since the pioneering work of Wahbe et al. [105]. This subsection introduces these techniques and their achievements as well as lessons learned by reviewing its development.

2.1.1 Original SFI

Wahbe et al. pioneered SFI techniques in the early 1990s to enhance the performance of communication-intensive computation modules, because the techniques place different modules in the same address space to reduce the cross-domain communication overhead [105]. They identified that indirect jumps and indirect stores are unsafe and must be confined. A verifier is deployed to check the validity of safeguarding dangerous operations before a module is executed, and this becomes standard practice in later SFI implementations. They identified two approaches to implementing SFI: (1) modifying a compiler back end such that the compiler directly emits the necessary sandboxing instructions for unsafe operations; (2) using binary rewriting to modify object code without changing a compiler. They implemented the first approach by modifying GCC for two reasons. First, leveraging a compiler makes it possible to directly utilize the optimizations that the compiler has to reduce the runtime overhead of sandboxing code. Second, binary rewriting, which they called binary patching, was not considered a mature technique at that time.

The goal of SFI is to confine an untrusted binary component inside its own segments: all indirect jumps go into a code segment, and all indirect stores access a

data segment. A segment is a contiguous memory space whose addresses have the same high-order bits, and these bits form the identifier of the segment. A dedicated register is used for all indirect stores: a store address with its high-order bits cleared is moved to the dedicated register, the high-order bits of the register are set to the identifier of the data segment, and the dedicated register is used for a store instruction. The segment identifier is also held in another dedicated register. Because the dedicated registers are reserved by the compiler, they can only be used by the sandboxing code. A similar strategy is used to confine indirect jumps. The entire sandboxing mechanism uses five dedicated registers: an additional register is used to hold the segment mask used to clear the high-order bits of an address.

In MIPS and Alpha machines, they achieved an average overhead lower than 5% for sandboxing store and jump instructions. The overhead for sandboxing load, store and jump instructions is around 18%–22%. They applied some optimizations to reduce the overhead, such as using a protection buffer surrounding segments and avoiding sandboxing certain addresses accessed through the stack pointer, besides applying some of the optimizations built in the compiler. The average overhead of reserving 5 registers out of 32 in MIPS machines is negligible: only 0.4%.

Unfortunately, they did not argue the correctness of their scheme and implementation formally.

2.1.2 SFI with Formal Verification

The above SFI mechanism works well for register-rich RISC architectures, but it becomes impractical on the x86 CISC architecture, because the latter has scarce general-purpose registers and variable-length instructions. Research efforts were made to develop SFI mechanisms for the x86 architecture in the late 1990s [29, 100], but later the designs were found flawed in their assumptions [66]. A lesson from those flawed mechanisms is that research communities begin to use formal methods to verify the security guarantee provided by a SFI design.

Abadi et al. advocated control flow integrity (CFI), which dictates that the execution of a program must follow a path that a reference CFG determines in advance [1].

They enforced CFI using a carefully designed technique: it inserts a unique identifier before a potential indirect jump target and checks the existence of the identifier before a corresponding jump instruction. The unique identifier does not exist anywhere in the code section, and the checking code uses a mangled version of the identifier to prevent the checking code itself from becoming a valid jump target. The overhead of enforcing the CFI ranges from 0% to 45% with an average of 16%.

In order to guarantee the correctness of their design, Abadi et al. applied formal methods to prove that the CFI design guarantees the enclosure of control flow transfers, based on semantics of a small instruction set that they developed [2]. The formal verification is conducted with pen and paper and checked by humans, and this means that the proof conductor must not make any mistakes in his or her reasoning. In addition, any implementation at the source and the object code level must be trusted, because the verification is only at the language level.

The security policy that CFI guarantees is much stronger than the segment-based confinement policy used in the original SFI, where control flow transfers are only limited into a specified code segment with no reference to a CFG. In principle, the strong policy may prevent any control-hijacking attack such as traditional stack overflow attack [3, 15] and newer return-oriented programming (ROP) attack [14, 98], because it prevents any control flow transfers that are not explicitly given in the reference CFG from being made.

Erlingsson et al. extended the CFI work into XFI: a fully fledged software-based access control system for executing untrusted binary modules [28]. XFI not only enforces control flow integrity but also controls memory accesses at any granularity with explicitly granted read, write and execute privileges. For indirect memory accesses, it uses a guard which checks the range of addresses accessed. A fast path can check the addresses in a single region; accesses to other regions are checked by using a slow path. XFI also uses a trusted verifier that statically examines the presence of checking code for memory accesses and control flow transfers. Unfortunately, it did not extend the formal methods used by CFI. XFI overhead varies significantly, e.g., the code size increase is 1.3–3.9 times, and the performance slowdown is 5%–93%

when using slow paths.

McCamant et al. designed another completely different approach to implementing SFI on the x86 architecture [66]. This approach divides the code section of a module into fixed-size chunks of 16 or 32 bytes. A legal jump target must start a new chunk, and no instructions may cross a chunk boundary. This is made possible by padding chunks with the `nop` instruction. Both dangerous instructions and their sandboxing instructions must be placed inside one chunk, so that they are executed as an atomic unit. This design makes it enough to reserve one register for holding the address of indirect stores and jumps. They also developed optimizations to reduce the overhead of sandboxing code, besides using some of the strategies of Wahbe et al.. For example, a segment starting at address zero is reserved, so that address masking can be done in one instruction instead of two. The runtime slowdown is about 21% on average, and the code size increase is about 62%–96%.

In order to argue the security properties of their design, McCamant used the ACL2 first-order logic proof assistant to verify it [64]. He formalized the constraints of the verifier and simulated a small subset of the x86 ISA; he proved that if code passed the verifier check, then it was confined properly.

Winwood et al. designed a sandboxing system to enforce control flow integrity by using lightweight binary rewriting and reference monitors, assisted by hardware-based memory projection mechanisms of the Alpha architecture [108]. The rewriting replaces every indirect jump with a direct jump to a jump monitor, which in turn jumps to a security monitor to check the validity of the jump against a given security policy. Both monitors are trusted, and a verifier is used to check that there are no indirect jumps and system calls in a rewritten program. In order to show that this design correctly enforces the control flow integrity, they used the Isabelle/HOL higher-order logic proof assistant to verify the safety guarantee of their design. They formalized the semantics of a small subset of the Alpha ISA and the semantics of the protection mechanisms; based on the semantics, they formally verified the security of their approach.

A common characteristic of these SFI formal verifications is that they prove the

security of a scheme at an abstracted language level. For example, if a scheme uses a security monitor or a verifier, the verification simply assumes the correctness of the monitor or the verifier and gives its semantics [64, 108]. Whether the model is faithful to its implementation is not addressed at all. However, it has been well known that modeling is one of the weakest links that can easily go silently wrong in formal verification [55]. A second common characteristic is that they all apply a self-defined formal semantics of a small subset of some ISA. Section 1.4.1 has already emphasized the importance of a realistic and faithful semantics.

2.1.3 Google Native Client

The Native Client project adopts McCamant's instruction bundling and padding approach to SFI with the addition of springboard techniques to confine system calls [4, 95, 111]. In modern architectures such as x86, x86-64, and ARM, it achieves an impressive average overhead less than 5%. Code review and testing are used to find bugs in verifier source code, and no formal methods are conducted to guarantee the correctness of the design and implementation.

2.2 Program Logic for Machine Code

Reasoning about programs formally dates back to McCarthy, Floyd and Hoare in 1960s, where either annotated flowcharts or formulas were used to describe program states mathematically [33, 46, 67]. This subsection reviews important development closely related to theorem proving on machine code programs.

2.2.1 Hoare-style Logic

Hoare's work was particularly influential in formal verification, because he showed an axiomatic program logic in which mathematical statements of a program could be systematically composed from predefined rules for individual statements in a language [46]. Since the invention of the LCF system by Robin Milner [69], a mechanized proof assistant has become a standard tool for conducting Hoare logic reasoning [40]. The basic principle of the reasoning process is the following: a logic judgment specifies a piece of code in the format of a triple: $\{P\}\ C\ \{Q\}$, where

P and Q are the precondition and postcondition of code C. It states that if the precondition P is true before the execution of C, then the postcondition Q is true when C terminates. The triple of a program is developed by applying inference rules to individual triples of the constructs of the program, and this process may be viewed as construction of a proof tree in a bottom-up fashion. Traditional inference rules include Sequencing, Strengthen, Weaken and so on [25].

The early Hoare logic is designed to reason about programs written in high-level languages which have well-defined constructs. One of its characteristics is that its inference rules must be developed based on the structures of the constructs. This often results in a complicated rule system. For example, in order to reason about the `while` loop, the logic requires proving a rule for the `while` structure [25]; in order to reason about function calls, the logic needs rules proven for them [93]. When research communities find interests in dealing with low-level language or machine-code programs, they have adopted the basic form of Hoare logic to different variations.

Myreen et al. developed a Hoare logic for machine code, which is very similar to the traditional Hoare logic, i.e., a complex rule system needs to be developed in order to deal with machine-level jumps [75, 76]. In order to address the reusability of proofs, Myreen et al. developed a decompiler approach to reasoning about machine code [77]. It converts machine-code programs into logic functions inside the HOL proof assistant. As a result, reasoning about programs becomes reasoning about functions in the theorem prover, and the latter is much easier and more efficient. Once programs written in different ISAs are decompiled into the logic system of HOL, proofs about functions can be reused. Based on this method, Myreen verified implementations of the Cheney garbage collector written in x86, ARM and PowerPC assembly languages, respectively [17, 73]. Furthermore, based on his machine-code Hoare logic, he successfully demonstrated a proven just-in-time compiler for x86 [74].

However, using a Hoare logic to reason about low-level or machine-code programs has some fundamental limitations. For example, a low-level language does not have structured constructs. The most common type of control flow transfers is unstructured or arbitrary jumps. The Hoare logic simply cannot handle them effectively.

To address this problem, Tan and Appel developed a compositional logic to reason about arbitrary control flow transfers in low-level code [103]. The logic uses the concept of label continuation, which is a pair of a program address—the label—and a state predicate. They interpreted the pair in a continuation style: a true predicate at a label means that it is safe to jump to that location. This interpretation allows them to reason about any types of jumps commonly seen in low-level language programs. However, in order to prove the soundness of the logic, they had to create a complex semantics, which in turn resulted in a complicated soundness proof. Using this logic, Tan not only proved the typing rules used in the typed assembly language for the foundational proof-carrying code project, but also proved a memory safety property in SPARC assembly code by encoding the safety property into the type system [102].

Benton proposed a typed, compositional logic for a stack-based abstract machine and established the soundness by adopting Tan and Appel's interpretation [8]. Saabas and Uustalu developed a natural semantics for an unstructured low-level language and showed that Hoare inference rules may be derived from this semantics [91].

2.2.2 Certified Assembly Programming

Inspired by proof-carrying code which mainly focuses on verification of type safety, Shao's group developed certified assembly programming (CAP) techniques to reason about the functional correctness of low-level programs [113]. The basic approach starts with a given specification, which consists of a collection of state predicates at different program locations. The process interprets each instruction of the program, comes up with intermediate assertions, and checks that the given specification is consistent against the semantics of instructions. This is very similar to Floyd's inductive assertion, which is also used by proof-carrying code and VCG-based approaches [79, 109]. CAP's contribution is formalizing this idea in a mechanized theorem prover and showing that it is possible to use it to reason about the functional correctness of low-level programs. In a later development, Shao's group extended CAP to a family of techniques which include modular reasoning [82, 83], stack-based control reasoning [31], concurrent program reasoning [114], and hardware interrupt

reasoning [30].

Compared to Hoare-style reasoning, the CAP family of techniques has a simpler structure and does not require complicated rule systems. Additionally, it does not depend on the termination of code, because its soundness is interpreted by induction on execution steps [113].

The disadvantage of CAP is that the reasoning process is interactive, requiring much manual work. In addition, it is posterior in the sense that when given a correct specification, it can verify its correctness. However, writing a correct specification for machine-code programs requires much more effort than verifying it, which is the rationale of proof-carrying code to separate the proof generation process from the proof verification process, so that a code user only needs to verify a proof [79]. It has not been shown that there exist effective ways to give correct specifications for low-level or machine-code programs.

2.2.3 Other Work

Boyer and Yu made the first attempt to verify real-world executables generated by GCC. They used the Boyer-Moore theorem prover Nqthm to prove the functional correctness of a series of small machine-code programs for the Motorola MC68020 microprocessor [13, 52]. Their proof is lengthy and labor intensive.

2.2.3.1 Verification Condition Generation

Dijkstra introduced the concept of the weakest preconditions in 1970s [24, 25], and the computation of the weakest preconditions lays a solid foundation for verification condition generation (VCG) [27, 53]. VCG verifies a program by generating a set of mathematical predicates about program variables at certain locations—so called verification conditions; if these verification conditions can be proven, then the original program is implied to be correct. This method has been studied extensively in the literature [20, 32, 48, 54, 59]. In practice, VCG has a large TCB, because it normally requires a custom-built verification condition generator, which is a large piece of software. The proof or discharge of verification conditions is usually conducted in a solver or theorem prover [109].

Matthews et al. proposed using a theorem prover to generate verification conditions based on an operational semantics of a machine language [63]. Hardin used this method to verify Rockwell Collins AAMP7G machine code [45], and the microcode of AAMP7 is formally verified [43].

2.2.3.2 Proof-carrying Code

Necula introduced the concept of proof-carrying code (PCC) in which proofs of certain properties of code can be attached into the code, so that a code user can easily verify that the code adheres to certain security policies [79, 80]. Its predominant application is verifying type safety of assembly code, which was advocated by Morrisett et al. [71]. A program is annotated with types according to typing rules during compilation, and a well-typed program implies that certain safety properties hold. The typing rules are defined directly in the PCC system. Tan proved the typing rules in the foundational PCC project based on instruction semantics [102].

Leroy showed that a PCC system assumes the existence of a certifying compiler which creates proofs that a code user can verify, or hints from which proofs can be constructed easily; in theory, a certifying compiler implies a certified compiler, and vice versa [60]. This means that a PCC system must implement some theorem proving work equivalent to that of a certified compiler, revealing the biggest obstacle of developing a practical PCC system: generating proofs, not verifying them. Leroy et al. successfully demonstrated that utilizing a mechanized proof assistant is a practical way to construct a certified compiler [11, 60].

2.2.3.3 Decompilation into Logic Functions

Li developed a decompilation strategy using the idea of state monads in his validated compilation work [61]. This method can convert very messy machine code with arbitrary jumps into a logic function, without requiring certain structures of the code to be discovered, which is a prerequisite in the previous decompilation method discussed in Section 2.2.1. Compared with the existing theorem proving methods discussed above, this approach is more promising, because it converts reasoning about arbitrary machine code into reasoning about functions in the logic system

of a mechanized proof assistant with no presumed structures on the machine code. The next challenge is to automate the reasoning process for logic functions, and there is much work ahead in this direction.

2.3 Summary

It can be seen from the development of SFI that existing work lacks rigorous verification, and this results in a large TCB in implementations; the TCB includes verifier source code and the compiler that translates the source code into the executable machine code. In parallel, theorem proving for verifying machine-code programs has been studied, but remains to be shown as a practical method to verify safety properties of binary programs. This dissertation shows that with properly developed verification tools, SFI implementation may achieve a very small TCB for embedded systems.

CHAPTER 3

THE ARMOR SFI IMPLEMENTATION

As discussed in Section 1.5.2, the high-confidence assurance that ARMor provides is due to formal verification using the HOL proof assistant. Verification is a static reasoning process in HOL. It is well known that statically determining the truth of a safety property is undecidable in general. Therefore, I implemented some binary transformations based on Diablo, and they add necessary safety checks to a program, making verifying isolation properties possible. The work presented in this chapter forms the processing step marked as "SFI transform" in Figure 1.2.

3.1 Motivation

This subsection uses concrete examples to show the necessity of inserting dynamic checking code and the invariants that the check provides.

3.1.1 Dangerous Indirect Stores

Suppose that we have a shorter binary program than the one shown in Figure 1.3(a), where the `foo` function does not have blk3. Recall that one of our goals is to verify memory safety, which requires showing that the store instruction `strb R1,[R2]` is safe, i.e., the address in the R2 register is in the region represented by the `mem` set given in Figure 1.3(b). Because register R2 is uninitialized in this program, its value can be any 32-bit pattern. The attempt to show that it is in the set of `mem` fails naturally.

However, when we place blk3 back into the program, we will be able to verify the memory safety. The block tests whether the value of R2 equals 0x40000000. If it is the expected address, then control goes to the store instruction; otherwise, control goes back to the caller. In the first case, we have a constraint about the value of register R2, which says that the value is 0x40000000, and now we can successfully

prove that the value of R2 is in the `mem` set. In the second case, we do not need to show that the store instruction is safe, because control does not go to the instruction.

A question is: Which indirect stores are not safe and require checking code to be inserted before them, to provide safety constraints? The ARM ISA is a load-store architecture, and strictly speaking, every store is indirect. However, inserting checking code before every store instruction is not necessary, because in some situations, the value of the target address can be determined statically in the logic system of the proof assistant. For example, a store instruction that uses a register whose value is previously set to a safe constant does not need additional checks. Generally speaking, store instructions whose addresses are uninitialized, derived from user input, or computed in a complex way need additional safeguards.

3.1.2 Dangerous Indirect Jumps

Jumps through registers or memory locations are dangerous for control flow integrity. For example, suppose there is a C program, whose source code and ARM assembly code generated by GCC are shown in Figure 3.1. Dangerous jumps happen in functions `gee` and `haa`. The multiword store instruction at line 7 of Figure 3.1(b) pushes the value of the link register R14 onto the stack, and R14 holds the return address of function `gee` when `gee` is called in `main` at line 19. The next three instructions set up the frame pointer R11, allocate space for the local variable `x`, and place the address of `x` in register R0, which is also the value of R13. The stack frame of function `gee` before it calls function `haa` is shown in Figure 3.2(a), where r11′ is the value of R11 before it is defined at line 8.

Throughout this dissertation, I use capital letters such as "R" and "PC" to refer to the names of registers such as register R11 or register PC, and lower case letters "r" and "pc" to refer to the values in the corresponding registers.

When function `haa` is called, the address at (r0+12) is set to 0 at line 3. Unfortunately, (r0+12) is the location where the return address r14 is stored for function `gee`. Figure 3.2(b) shows the same stack frame after function `haa` returns to `gee`. The last instruction of `gee` at line 14 loads values at addresses (r11-4), (r11-8), and (r11-12)

```
void haa(int *pi) {
  *(pi + 3) = 0;
}

int gee() {
  int x;
  haa(&x);
  return x + 1;
}

int main() {
  gee();
  return 0;
}
```

(a) C code

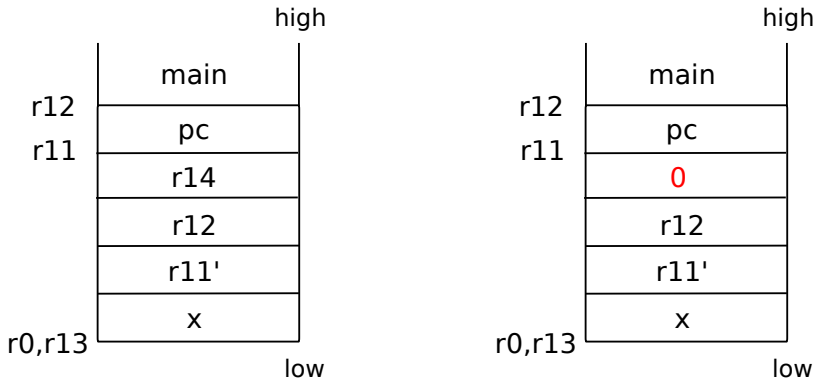
```
1   <haa>
2   mov    R3,#0
3   str    R3,[R0,#12]
4   mov    PC,R14

5   <gee>
6   mov    R12,R13
7   stmdb  R13!,{R11,R12,R14,PC}
8   sub    R11,R12,#4
9   sub    R13,R13,#4
10  sub    R0,R11,#16
11  bl     <haa>
12  ldr    R0,[R11,#-16]
13  add    R0,R0,#1
14  ldmdb  R11,{R11,R13,PC}

15  <main>
16  mov    R12,R13
17  stmdb  R13!,{R11,R12,R14,PC}
18  sub    R11,R12,#4
19  bl     <gee>
20  mov    R0,#0
21  ldmdb  R11,{R11,R13,PC}
```

(b) Compiled ARM assembly code

Figure 3.1: A program with a dangerous jump



(a) Stack before line 11

(b) Return address overwritten

Figure 3.2: Compromising control flow integrity

into registers PC, R13, and R11, respectively. Now PC is assigned the value 0, and when the next instruction is executed, unpredictable behavior occurs. In embedded systems which do not have isolation support, a hardware trap may crash the entire system.

In order to provide isolation, these dangerous uses of addresses must be controlled such that a faulty computation cannot affect other innocent computations.

3.2 ARMor's SFI Mechanisms

This subsection discusses the binary transformations implemented in ARMor to support the isolation service in embedded systems.

3.2.1 Checking Unknown Store Addresses

A basic idea of preventing dangerous store instructions from accessing unauthorized memory addresses is inserting checking code before the instructions whose address can not be determined statically in the logic system. The inserted code checks the validity of the address range of a store instruction against the address regions given in the security policy. If the range falls within the given regions, then the checking code allows the store instruction to execute. Otherwise, the checking code aborts the current computation. Figure 3.3 illustrates this transformation with an example. The original code is a multiple-store instruction, which saves the values

```
stmia  R2,{R6,R7}
```

(a) Original code

```
mov    R0,R2
add    R1,R0,#0x8
bl     dguard
stmia  R2,{R6,R7}
```

(b) Rewritten code

```
if [r0, r1) falls in the global data section, the stack, or I/O region then
    return
else
    abort
```

(c) Pseudo-code of the checking routine `dguard`

Figure 3.3: Checking unknown store addresses

of registers R6 and R7 (8 bytes in total) at addresses pointed to by register R2: [r2, r2+8).

Constraints in embedded systems and the primary research goals of this dissertation shape the prototype design presented here. First, a program in an embedded system often needs to access different memory regions. As a result, I use three disjoint regions in the security policy: the global data section, the stack, and the I/O addresses. Although this choice bring flexibility in memory access, it impacts performance adversely. For example, the checking routine `dguard` in Figure 3.3(c) needs to compare an address range with three different regions, and this is a very expensive operation. Second, one of the major research goals of this dissertation is to explore the influence of a complex checking routine on formal verification. If it is effective to reason about complicated checking routines formally, then it is relatively easy to adopt the formal method to reason about simple routines. Third, embedded systems usually have very limited memory. This leads to a fine-grained memory control policy. Specifically, the memory regions are specified at arbitrary address boundaries, and the boundaries do not have to be a power of two. In addition, there are no protection buffers around data regions. Otherwise, limited memory in an embedded system would become tighter. Nor is the `dguard` function inlined, because the function body is not short—it has 18 instructions. Inlining would increase the code size of a rewritten program too much.

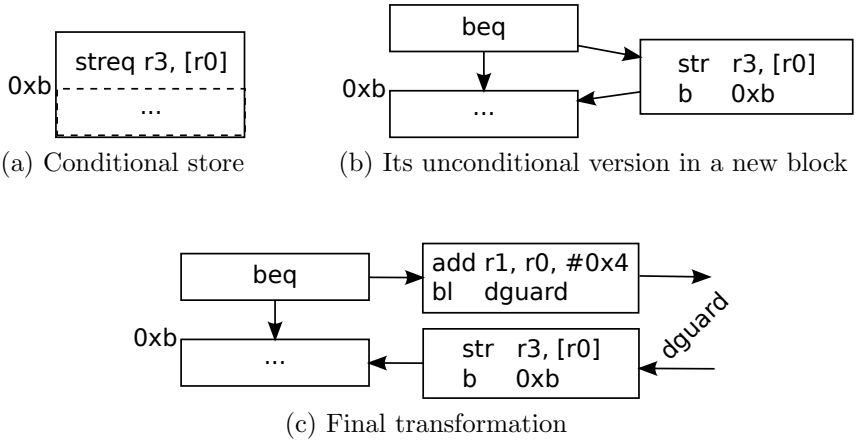
Given a different system configuration which does not have the above memory limitations, a completely different design choice may be used. For example, masking operations are a natural selection to boost performance for coarse-grained segment-based memory policy in a memory rich system, as discussed in Section 1.2. I will discuss alternative options in detail in Section 3.3.

3.2.1.1 Checking Conditional Store Instructions

The ARM ISA has a distinctive feature of allowing almost every instruction to execute conditionally [7]. There is a condition field in the instruction encoding, and the value of the field indicates one of the 14 available conditions such as equality test

results. A conditional instruction is indicated by conditional code in assembly. For instance, the "eq" suffix in Figure 3.4 is the conditional code for an equality test. If the status flags of the processor indicate that the corresponding condition is true when the instruction starts executing, then the instruction executes normally. Otherwise, the instruction does nothing, just like a `nop` instruction.

Conditional store instructions make the above checking scheme fail, because the status flags may be changed by the address checking routine. In order to keep the correct status flags for a conditional store instruction, the original flags must be preserved before and restored after the checking routine executes. In addition, the instructions that call the checking function must be made conditional, in order to keep the original semantics of the program. It is not straightforward to make all the flag operations correct, and a simple method is designed to convert a conditional store to its unconditional version while keeping the same semantics of the original code. The method splits the basic block containing a conditional store instruction into two at the location right after the store instruction, makes a unconditional copy of the store instruction in a new empty basic block, and replaces the conditional store instruction with a conditional branch instruction which jumps to the new block and has the same condition code as the original store instruction. The new block unconditionally jumps back to the instruction right after the splitting point. Afterwards, the checking routine is inserted for the unconditional version of the store instruction like before.



(a) Conditional store

(b) Its unconditional version in a new block

(c) Final transformation

Figure 3.4: Safeguarding a conditional store instruction

Figure 3.4 shows this special transformation, where solid lines represent basic blocks, dashed lines are inside basic blocks, and arrows are control flow transfers.

3.2.2 Protecting Return Addresses

Section 3.1.2 illustrated the danger of overwriting a return address stored on the stack. There are several possible solutions to this issue. For example, one can remember the location where the return address is saved and check if any store address range includes that location. Because there are many return addresses spilled on the stack, all of them must be checked separately. This would slow down pointer operations too much. In typed high-level or assembly languages, a type can be defined to prevent out-of-bound memory accesses [5,71], but there is no such type information at the machine-code level. Dynamic binary instrumentation records a shadow value for every memory address, which can dynamically determine out-of-bound memory accesses [81]. This requires a huge amount of memory that small embedded systems do not have.

There is another method: separating control from data. Return addresses are control information related to the CFG of a program, which the program should never directly modify for the sake of control flow integrity, while the stack contains local variables which the program must be able to modify. I separate the control information from the stack data by introducing another fixed memory region, as shown in Figure 3.5. It is called the *control stack*: it is used to save return addresses. When a function is called, this transformation saves a copy of the return address onto the control stack; when the function returns, the transformation assigns the PC with

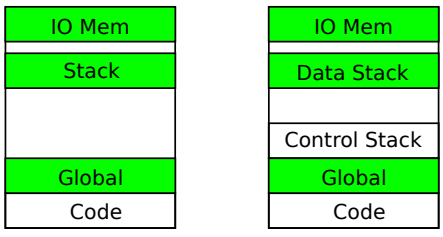


Figure 3.5: Control stack. Left: the original memory layout of an executable. Right: ARMor's transformed layout.

the address taken from the control stack. GCC is patched to reserve a dedicated register, R8, as the control stack pointer. The original stack is not changed and is now called the *data stack*. This transformation leaves the data stack as it is, allowing out-of-bound memory accesses on it. However, because no control information is used from the data stack, the integrity of function returns is maintained.

An issue this transformation introduces is whether the store instruction that pushes a return address onto the control stack needs to be sandboxed, because it is an indirect store. ARMor's formal verification showed that for nonrecursive functions, no checking code is needed, and the verification may prove the memory safety of the control stack pointer without additional constraints.

The space overhead of the control stack is low, because each function call takes an additional four bytes. The total space cost is four times the maximum depth of nested function calls. Another overhead is the reservation of the control stack pointer. Previous research shows that performance slowdown of reserving 5 registers out of 32 is negligible [105]. Even in the register-scarce x86 architecture, reserving a single register does not incur obvious overhead [28]. The ARM architecture has 15 registers; the influence of reserving one register should also be negligible, judging from these existing results.

3.2.3 Constraining Indirect Jumps

Besides function returns, other types of indirect jumps are jump tables and function pointers. I enforce the integrity of these indirect control flow transfers by a different method, since they do not have the stack-based control characteristics in a binary program.

The technique is adopted from the work of Abadi et al. [1] and illustrated in Figure 3.6. This transformation inserts a unique identifier that is not present in the code of a program before each potential jump target. In the example, the unique identifier for the target is 0x8. Before a corresponding jumping instruction, additional code is inserted to check the presence of the identifier, as shown in Figure 3.6(b). The checking method is the following: a fixed operation is used to restore the

```
        mov PC, R2          ;indirect jump
        ...
target:
        str R1, [R0]        ;target
```

(a) Original code

```
        mov  R0, 0x10       ;load mangled word pattern
        mov  R0, R0, ROR #1 ;right rotate 1 bit
        ldr  R1, [R2, #-4]  ;R1 <- Mem[R2 - 4]
        cmp  R0, R1         ;compare ids
        bne  invalid        ;abort if not equal
        mov  PC, R2         ;indirect jump
        ...
        0x8                 ;unique word pattern
target:
        str R1, [R0]        ;target
```

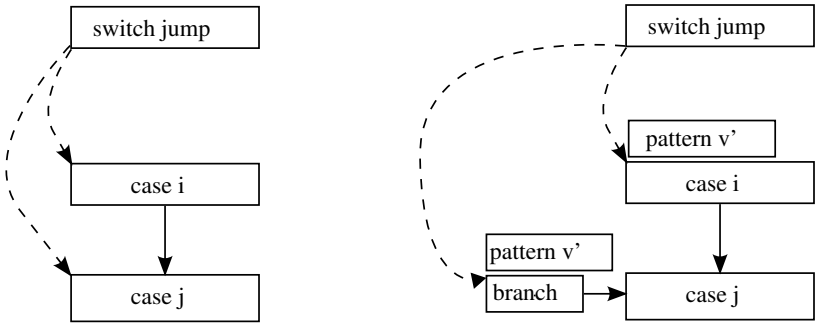
(b) A unique ID and the check for its presence

Figure 3.6: Constraining unknown jumps

unique identifier from a mangled value of the identifier. A correct restoration of the unique identifier implies a valid target, and an incorrect restored value implies that the jump target is illegal, because a modified jump target is not proceeded by the unique identifier. The fixed operation used here is right rotation by one, and the corresponding mangled value is left rotation by one. A mangled value of the unique identifier is used to maintain the uniqueness of the target such that the checking code itself does not become a valid jump target.

In the ARM architecture, a unique identifier can be placed in a data pool. A data pool is addresses in the code section storing constant data [7].

Care must be taken when inserting a unique identifier before the jump target that has an incoming fall-through edge, because when an identifier is inserted before that target, the fall-through execution may cause an instruction decoding error. A solution is inserting a new basic block with a branch instruction that jumps to the target and treating the branch instruction as the jump target, so that the fall-through relationship is correctly maintained. Figure 3.7 shows the structural change for a



(a) Original target (b) Modified target

Figure 3.7: Constraining a jump table with a fall-through edge

jump table with a fall-through edge, where dashed arrows represent indirect jumps, and solid arrows are direct jumps.

3.3 Discussion of Related Work

Section 1.2 introduced the two central issues in SFI implementation: efficiency and correctness. The efficiency issue is closely related to the security policy that an SFI mechanism enforces: the less strict the policy, the less overhead the mechanism incurs.

For memory safety, there are two extreme policies: segment-based policy and fine-grained policy. Segment-based policy only requires all store instructions to access a single contiguous memory region, this is enforced in the original SFI [105], PittsField [65], and Native Client [95]. The corresponding enforcement mechanisms can be implemented by masking store addresses: setting the high-order bits of an address to the specified segment. This is very efficient, because masking can be done in two instructions. To further reduce overhead, the addresses below the data segment can be reserved empty such that the masking can be done in one instruction. In contrast, a fine-grained memory policy can specify a memory region at any location with certain privileges, and a program may have multiple regions each with different privileges. A representative implementation supporting this policy is XFI [28].

ARMor's memory safety policy is in between these two extreme cases. It allows multiple flexibly specified regions similar to the fine-grained policy, but it does not enforce specific read, write, or execute privileges for its isolation purpose. This policy

is chosen due to the limitations of memory in embedded systems and its research goals stated in Section 3.2.1.

A similar situation exists for control flow transfer policies. A very loose policy may only require that all jumps are sandboxed to a single contiguous code region, and this is enforced in the original SFI [105]. A slightly stricter policy may demand that jump targets are only at fixed locations in the code section, such as the one implemented in PittsField [65], which divides the code section into fixed-size chunks, and jumps can only target at the start of chunks. In fact, this is necessary to guarantee that address masking code cannot be skipped over store instructions. The strictest policy demands that any jump must follow a path in a given CFG determined in advance. This is enforced by CFI and XFI [1, 28]. I also used this strictest policy for the control flow integrity in ARMor, because only this policy prevents the return-oriented programming attacks [14, 98]. Other policies need additional mechanisms to prevent such attacks, such as trampoline/springboard techniques or limiting addressing modes and pointer uses [95].

Erlingsson et al. used two stacks for each untrusted module in XFI: a scoped stack and an allocation stack [28]. The scoped stack stores function return addresses and local variables, and it can only be accessed through a fixed offset from its stack pointer. Because of this invariant, checks for several memory accesses may be optimized into a single check. The allocation stack stores values accessed through pointers. Both stack pointers are checked to maintain their validity before they are set to new values.

Inspired by XFI, I introduced the control stack in ARMor. At the machine-code level where ARMor operates, there is no reliable way to distinguish local variables whose addresses are taken from those whose addresses are not taken. Therefore, the control stack only stores return addresses. In addition, no checks are needed for the control stack pointer for nonrecursive function calls due to the formal verification of ARMor.

The correctness issue of SFI design and implementation has not been sufficiently addressed as mentioned in Section 1.2. I postpone its discussion to Section 8.1 after presenting ARMor's verification.

CHAPTER 4

$\mathcal{L}_{fn}$: A PROGRAM LOGIC

The theoretical foundation of ARMor's formal verification is $\mathcal{L}_{fn}$: a novel program logic that facilitates proof automation. I developed this logic to address limitations of Hoare logic and the interactive nature of higher-order logic theorem proving in verifying shallow safety properties.

4.1 Motivation

A program logic plays a central role in reasoning about programs in a mechanized theorem prover.[1] Section 2.2.1 introduced the Hoare logic and its adoption in reasoning about binary programs. However, there are fundamental limitations in Hoare-style logics that make them unsuitable for low-level or machine-code programs. First, a Hoare logic is structured according to the constructs of a language. Each structure requires certain rules to be proven in the logic in order to compose a code judgment for that construct, e.g., one of the most famous structures and its rule are the `while` loop and the While rule [25, 41]. However, machine-code programs do not naturally have such structures. As a result, although a Hoare logic may be developed for reasoning about binary programs, it can only deal with certain code whose structures are obvious such as function calls or whose structures can be heuristically rebuilt such as simple loops [76, 77]. For most arbitrary jumps commonly seen in machine-code programs, it is extremely difficult to develop rules for them. For example, there are no standard rules for jumps such as those shown in Figure 4.1.

---

[1] By convention, when a program logic is developed inside the logic system of a proof assistant, the latter is called metalogic to distinguish the target logic from the existing logic environment of the proof assistant. In my case, $\mathcal{L}_{fn}$ is the target logic, and the logic system of HOL is the metalogic.

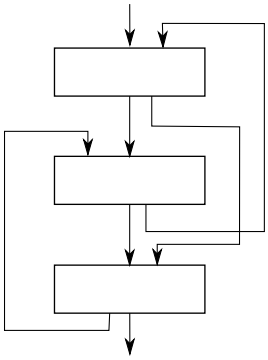


Figure 4.1: Unstructured jumps

Second, a Hoare logic is unable to reason about infinite loops. There are total and partial specifications in a Hoare logic [25, 93]. The total specification states that if the precondition is true and the code terminates, then the postcondition is true. The partial specification does not require the termination condition. However, in practice, either specification is not helpful in reasoning about infinite loops, because the postconditions of the loops cannot be verified. In embedded systems, many control loops are purposely written to iterate forever.

Third, a Hoare logic only specifies the precondition and postcondition of code, and this means that only the prestate and poststate are asserted in the specification. Intermediate states are completely suppressed in a Hoare judgment. This is a problem for verifying safety properties, because safety verification requires that every state of a program should be validated. For example, assume that a piece of code accesses illegal memory locations but later on restores their values. One may use a Hoare logic to show that the ending state of the code stays the same as the starting state and to argue that no security violations occur.

As an alternative to Hoare-style reasoning, Section 2.2.2 introduced certified assembly programming. There are also fundamental issues that hamper its practicality. First, how to efficiently write correct specifications for low-level or binary programs? Manually working on assembly code is very inefficient and error-prone, and there are no good methods to address this issue. Second, the proof is interactive. Depending on experts to conduct proofs at machine-code level is just noneffective. In order to prove

safety properties, this method requires interactive verification at each instruction.

$\mathcal{L}_{fn}$ is designed to address the issues that a Hoare logic has. Based on it, I developed proof automation strategies for verifying shallow safety properties in a higher-order logic proof assistant. This chapter describes the logic, and Chapter 6 presents proof automation.

4.2 Overview

$\mathcal{L}_{fn}$ has three layers: the bottom layer is a parametric instruction semantics, the middle layer is a Hoare logic, and the top layer is a hierarchical function judgment. The parametric instruction semantics allows the logic to use an existing axiomatic semantics flexibly; it interprets the meaning of individual state transitions at the instruction level. The Hoare logic reasons about code blocks. A *code block* is a collection of instructions that has a single entry and one or more exits. The result of reasoning at this layer is Hoare judgments of code blocks. The hierarchical function judgment is the core of $\mathcal{L}_{fn}$, and it defines the meaning of a program. A function is a collection of code blocks, which has a single entry block and multiple exit blocks. It may or may not correspond to a function in a high-level language, depending on the convenience of verification. A program is a collection of functions organized as a tree structure similar to a call graph. Each function has its own judgment, and the judgment of the top-level or entry function is the judgment of the program.

4.3 Background: ARM Semantics

The bottom layer of $\mathcal{L}_{fn}$ is the instruction semantics. This subsection describes the existing ARM semantics mentioned in Section 1.4.2. It comes in as proven Hoare triples, based on a operational semantics, in the HOL theorem prover [37,75], but I use it as if it were an axiomatic semantics, because I take it for granted. Figure 4.2 shows the semantics for the store instruction: `strb R1,[R2]`. It says that after execution of the instruction, the value at memory address r2 is updated to the least significant byte of r1 (`w2w` converts a 32-bit word into an 8-bit word), and the PC is increased by 4. This semantics has some important properties which are summarized below. For a full treatment, interested readers may refer to the references mentioned above.

4.3.1 Code Assertion

The pair $(p, \texttt{0xE5C21000})$ is the code assertion for the instruction, meaning that the value `0xE5C21000` is stored at some memory address p. The value p can be thought as symbolic, and in fact, all nonconstant values in italic font are symbolic. In logic terminology, these symbolic values are free variables or universally quantified variables in the theorem.

4.3.2 Machine State Assertion

Machine states include registers, memory cells, status flags, and the current program status register. For example, `PC` p in the precondition of Figure 4.2 asserts that the program counter has value p and that p is word-aligned; `R` 2 $r2$ and `R` 1 $r1$ assert that registers R2 and R1 have values $r2$ and $r1$, respectively; `MEMORY` dom f asserts that some set of memory addresses dom has value f. The $\mapsto$ operator is defined as: $(a \mapsto b)\ f = \lambda x.(\text{ if } x = a \text{ then } b \text{ else } f\ x)$, namely, the result is a new function where a is mapped to b while other values stay unchanged. $((r2 \mapsto (\texttt{w2w}\ r1))\ f)$ means that only the value at address $r2$ is updated to $(\texttt{w2w}\ r1)$. Other machine state assertions include `S t` v: one of the status flags t (carry `sC`, negative `sN`, overflow `sV`, or zero `sZ`) has value v; `CPSR` x: the program status register, `CPSR`, has value x.

4.3.3 Separating Conjunction

The $*$ operator is the separating conjunction, and it has the expected properties as described in separation logic [90]: (1) a triple only asserts the local state, which is the parts of state that are used by the instruction, and a global version may be achieved by using the Frame rule, which adds resource assertions not used by a judgment onto it; (2) if a separating conjunction expression asserts a machine resource more than

$$\{\texttt{PC}\ p * \texttt{R}\ 2\ r2 * \texttt{R}\ 1\ r1 * \texttt{MEMORY}\ dom\ f * \langle r2 \in dom \rangle\}$$
$$(p, \texttt{0xE5C21000})\ //\ \texttt{strb R1,[R2]}$$
$$\{\texttt{PC}\ (p+4) * \texttt{R}\ 2\ r2 * \texttt{R}\ 1\ r1 * \texttt{MEMORY}\ dom\ ((r2 \mapsto (\texttt{w2w}\ r1))\ f)\}$$

Figure 4.2: Axiomatic semantics of `strb R1,[R2]`

once, excluding a pure assertion, then its value is `false`.

4.3.4 Pure Assertion

A pair of angled brackets, $\langle\rangle$, encloses a pure assertion, i.e., it does not assert any machine resource but serves as a predicate to specify some boolean relationship among logic variables and constants [75, 90]. Condition $\langle r2 \in dom\rangle$ states that $r2$ has to be in the domain of the memory function f in order for this transition to take place.

4.3.5 Lifted Operators

Some boolean operators such as implication ($\Rightarrow$) and disjunction ($\vee$) are lifted to the separating conjunction level. For example, $p \overset{*}{\Rightarrow} q$ means $\forall s.\ (p\ s \Rightarrow q\ s)$, and $p \overset{*}{\vee} q$ is $\lambda s.\ (p\ s \vee q\ s)$.

4.3.6 Conditional Execution

The semantics gives two separate theorems for an instruction with conditional execution, with each corresponding to a different condition. For instance, the conditional branch instruction ending blk4 in Figure 1.3(a), `bne foo`, has two theorems:

$$\begin{array}{ll} \{\texttt{PC}\ (p + 0x28) * \texttt{S}\ \texttt{sZ}\ z * \langle \neg z\rangle\} & \{\texttt{PC}\ (p + 0x28) * \texttt{S}\ \texttt{sZ}\ z * \langle z\rangle\} \\ (0x28, \texttt{0x1AFFFFF9}) & (0x28, \texttt{0x1AFFFFF9})\ //\text{bne foo} \\ \{\texttt{PC}\ (p + 0x14) * \texttt{S}\ \texttt{sZ}\ z\} & \{\texttt{PC}\ (p + 0x2C) * \texttt{S}\ \texttt{sZ}\ z\} \end{array} \quad (4.1)$$

where $\neg$ is the boolean negation. The left theorem describes the state transition when the condition is true, i.e., not equal (`ne`), and the right gives the state transition when the condition is false.

4.3.7 Additional Enhancements

I have developed additional assertions and theorems for the existing semantics during the development of ARMor, and one of them is an aggregated register assertion `REG`. An assertion `REG` *rf* means that the values of registers R0-R7 and R9-R14 are asserted by function *rf*. Registers R8 and R15 are excluded, because R8 is used as

the control stack pointer in ARMor (Section 3.2.2) and R15 is the program counter register in the ARM ISA.

Throughout the rest of this dissertation, I will use both the aggregated register assertion `REG` and the individual register assertion `R` depending on the convenience of description.

4.4 Assertion Language

The assertion language in $\mathcal{L}_{fn}$ is a set of label predicates embedded shallowly in the metalogic. Informally, *a label predicate* is a pair of a label (an instruction address) and a predicate, meaning that the predicate holds at the associated label. A set of label predicates means that there is a true label predicate in the set. Formally, the syntax of a label predicate is

$$\begin{array}{lcll} lp & \in & \mathit{LabelPred} & = & \mathit{LabelExp} \times \mathit{StateAssert} \\ l & \in & \mathit{LabelExp} & = & \text{word32} \\ p & \in & \mathit{StateAssert} & = & \text{separating conjunction expression.} \end{array}$$

Its interpretation is defined by a semantic function `LP2SP`, and another function, `LPSET`, interprets a set of label predicates:

$$\texttt{LP2SP}\ (l,\ p) = \texttt{PC}\ l\ * p$$

$$\texttt{LPSET}\ P\ =\ \lambda s.\ (\exists lp.\ lp \in P \wedge (\texttt{LP2SP}\ \ lp)\ s)\,.$$

Symbol $\stackrel{lp}{\Rightarrow}$ is used to denote the subsumption relation between two sets of label predicates and defined as:

$$P \stackrel{lp}{\Rightarrow} Q \textbf{ iff } \quad (\texttt{LPSET}\ P) \stackrel{*}{\Rightarrow} (\texttt{LPSET}\ Q).$$

In order to use the existing semantics in $\mathcal{L}_{fn}$, I define the following Ins rule:

$$\frac{\{\texttt{PC}\ l * p\}\ \mathit{ins}\ \{\texttt{PC}\ l' * q\}}{\texttt{ARM\_INS}\ (l,\ p)\ \mathit{ins}\ (l',\ q)}\ \text{Ins.}$$

The semantics `ARM_INS` is defined as an inductive relation with only one rule, Section 1.4.3. The purpose of this definition is to convert an instruction theorem into another that uses the label predicate as the assertion language. The reason behind

this conversion is that it is easy for the metalogic to operate on a pair, but it is difficult to operate on the $*$ operator, because the latter is not a constructor, and the metalogic cannot match an expression against it. Later when defining the function judgment of $\mathcal{L}_{fn}$, I take use of the matching ability of the metalogic on a pair.

4.5 A Hoare Logic

Hoare logic is the middle layer of $\mathcal{L}_{fn}$, and its purpose is to reason about a code block that only has sequential control flow transfers. I implemented it by the following set of definitions that bridges the gap between the underlying instruction transition and a Hoare judgment.

First, a `step` relation implements a state transition in the logic:

$$\texttt{step}\ ir\ i\ s\ t\ \textbf{iff}$$
$$\exists lp\ kq.\ (ir\ lp\ i\ kq) \wedge (\texttt{LP2SP}\ lp)\ s \wedge (\texttt{LP2SP}\ kq)\ t.$$

Informally, it says that a transition from state s to state t by instruction i under a given semantics ir is equivalent to a transition from s to t made by the instruction under the semantics. The first parameter, ir, is a relation of instruction transition, namely, an instruction semantics. It gives the logic the flexibility to use an existing axiomatic semantics in the format of a Hoare triple. For example, it can be the instruction semantics defined above (`ARM_INS`), or an augmented version for proving safety properties (to be described later in Chapter 6). For easy understanding, a reader can simply think of it as the `ARM_INS` relation for now. This definition is necessary, because it allows explicit references to states such as state s in the following definitions, instead of referring state assertions such as a label predicate lp.

Next, a sequencing relation implements the concept of n-step execution:

$$\begin{aligned}&\texttt{seq}\ ir\ C\ sq\ s\ \textbf{iff}\\ &\quad (sq\ 0 = s) \wedge\\ &\quad (\forall n.\ \texttt{if}\ \exists i \in C.\ \exists t.\ \texttt{step}\ ir\ i\ (sq\ n)\ t\\ &\qquad \texttt{then}\ \exists i \in C.\ \texttt{step}\ ir\ i\ (sq\ n)\ (sq\ (n+1))\\ &\qquad \texttt{else}\ sq\ (n+1) = sq\ n)\end{aligned}$$

where C is a set of instructions, and sq is a mapping from integer to state, which numbers states sequentially starting with 0. The definition specifies that in an instruction set, if there exists an instruction that can take a current state to the next, then just transition the current state; otherwise, execution gets stuck on a state.

With the above preparations, I next define a single-entry single-exit Hoare judgment and extend it to a multiple-entry multiple-exit Hoare judgment. The single-entry single-exit Hoare judgment is

$$\begin{aligned}&\texttt{sglspec}\ ir\ \{lp\}\ C\ \{kq\}\ \textbf{iff}\\ &\qquad \forall r\ s.\ ((\texttt{LP2SP}\ lp) * r)\ \ s \Rightarrow \forall sq.\ \texttt{seq}\ ir\ C\ sq\ s\\ &\qquad\qquad \Rightarrow \exists k.\ ((\texttt{LP2SP}\ kq) * r)\ \ (sq\ k).\end{aligned}$$

It reads that if the precondition lp holds for an initial state s, then k steps later, the postcondition kq holds for another state $(sq\ k)$. The universally quantified r forces any resources used by the code to be included in the pre- and postconditions.

The multiple-entry multiple-exit Hoare judgment is

$$\texttt{SPEC}\ ir\ \{P\}\ C\ \{Q\}\ \textbf{iff}\ \ \texttt{sglspec}\ ir\ (\texttt{LPSET}\ P)\ C\ (\texttt{LPSET}\ Q)$$

where P and Q are sets of label predicates, and ir is the parametric instruction semantics described above. It states that if there exists a true label predicate in the precondition, then there exists a true label predicate in the postcondition some steps later.

4.5.1 Inference Rules

From the above definitions, I proved basic inference rules about label predicates and Hoare judgments, some of which are listed in Figure 4.3, where the leading logic constant SPEC is omitted except for the Ins2Spec rule. The Ins2Spec rule does nothing more than getting instruction rules in this Hoare logic. Because this Hoare logic does not have its own built-in state transitions and relies on an existing axiomatic semantics represented by the ir parameter, this rule simply maps a state transition encoded in

Instruction rule:

$$\frac{ir\ \{p\}\ ins\ \{q\}}{\texttt{SPEC}\ ir\ \{p\}\ ins\ \{q\}}\ \text{Ins2Spec}$$

Label predicate rules:

$$\frac{}{P \overset{lp}{\Rightarrow} P}\ \text{LPRefl} \qquad \frac{P \overset{lp}{\Rightarrow} Q \qquad Q \overset{lp}{\Rightarrow} R}{P \overset{lp}{\Rightarrow} R}\ \text{LPTrans}$$

$$\frac{}{P \overset{lp}{\Rightarrow} (P \cup Q)}\text{LPExt} \qquad \frac{r \overset{*}{\Rightarrow} p}{(P \cup \{(l,r)\}) \overset{lp}{\Rightarrow} (P \cup \{(l,p)\})}\ \text{Imp2LPimp}$$

Hoare rules:

$$\frac{ir\ \{P1\}\ C1\ \{Q1\} \qquad ir\ \{P2\}\ C2\ \{Q2\}}{ir\ \{P1 \cup P2\}\ (C1 \cup C2)\ \{Q1 \cup Q2\}}\ \text{Union}$$

$$\frac{ir\ \{P \cup M\}\ C\ \{Q \cup M\} \qquad ir\ \{M\}\ C\ \{Q\}}{ir\ \{P \cup M\}\ C\ \{Q\}}\ \text{Discharge}$$

$$\frac{ir\ \{P\}\ C1\ \{M\} \qquad ir\ \{M\}\ C2\ \{Q\}}{ir\ \{P\}\ \{C1 \cup C2\}\ \{Q\}}\ \text{Sequence}$$

$$\frac{ir\ \{P\}\ C\ \{Q\}}{ir\ \{(l, p * r) | (l,p) \in P\}\ C\ \{(k, q * r) | (k,q) \in Q\}}\ \text{Frame}$$

$$\frac{ir\ \{P\}\ C\ \{Q\} \qquad R \overset{lp}{\Rightarrow} P}{ir\ \{R\}\ C\ \{Q\}}\ \text{Strengthen}$$

$$\frac{ir\ \{P\}\ C\ \{Q\} \qquad Q \overset{lp}{\Rightarrow} R}{ir\ \{P\}\ C\ \{R\}}\ \text{Weaken}$$

LPMerge:

$$ir\ \{P \cup \{(l,p)\} \cup \{(l,q)\}\}\ C\ \{Q\} = ir\ \{P \cup \{(l, p \veebar q)\}\}\ C\ \{Q\}$$
$$ir\ \{P\}\ C\ \{Q \cup \{(l,p)\} \cup \{(l,q)\}\} = ir\ \{P\}\ C\ \{Q \cup \{(l, p \veebar q)\}\}$$

Figure 4.3: Proven inference rules

ir to an instruction rule in the Hoare logic. The Imp2LPimp rule makes it easier to use and derive condition subsumption relations, and the LPExt rule is very useful in strengthening or weakening a Hoare judgment.

The Union rule composes the judgments of small pieces of code into a single "bigger" judgment, and is used to derive other rules such as the Sequence rule. The Discharge rule removes unnecessary intermediate label predicate entries in the

postcondition. The Sequence rule is used to construct the judgment for basic blocks. Weaken and Strengthen are used to change the post- and preconditions of a Hoare judgment. The Frame rule extends a local judgment to the global version. Label predicate merge rules are used to merge and split label predicate entries.

These rules are standard inference rules in a traditional Hoare logic and similar to some of the rules developed in existing work [75, 103]. A major difference is that there are no loop rules or call rules to compose loops or function calls in this logic—in fact, there are no compositional rules for structures other than simple sequential code blocks, because these structures are reasoned about in the top layer of the logic without using any compositional rules.

4.5.2 Automatic Composition of Code Block Judgment

Unlike in a traditional Hoare logic, the role of this Hoare logic is very limited in $\mathcal{L}_{fn}$, and it is only used to compose judgments for code blocks. There are at least two different ways to compose a code block judgment from smaller judgments in this Hoare logic. One is using the traditional Sequence rule, and this has been well studied [40, 46]. This section describes an alternative process to illustrate how to use some of the rules. The inferences rules used in this process are Ins2Spec, Frame, Union, Discharge, Strengthen, and LPExt.

Let me take the code block, blk4, of Figure 1.3(a) as an example and compose its Hoare judgments. For a complete description of the Hoare-style reasoning process, I start with a preparation step which generates the instruction rules. This step applies the Ins rule defined in Section 4.4 to an existing instruction axiom to obtain a corresponding `ARM_INS` relation. Next, the Ins2Spec rule is used on the relation to derive the `SPEC ARM_INS` Hoare judgment for the instruction, where the semantic parameter *ir* is instantiated with `ARM_INS`. After applying this step to the existing semantics shown in Figure 4.2, we obtain the following Hoare rule for the same store instruction:[2]

[2]A Hoare triple is commonly written as {P} C {Q} in literature. Here, P, C and Q are sets, whose content is also written in braces by convention. For clarity, I only use one pair of braces in writing pre- and postconditions and do not use braces for code.

$$\begin{aligned}&\texttt{SPEC ARM\_INS}\ \{(0x20, \texttt{MEMORY}\ dom\ f * \langle r2 \in dom\rangle * \texttt{R}\ 2\ r2 * \texttt{R}\ 1\ r1)\}\\ &\qquad (0x20, \texttt{0xE5C21000})\ //\ \text{strb R1, [R2]}\\ &\qquad \{(0x24, \texttt{MEMORY}\ dom\ ((r2 \mapsto (\texttt{w2w}\ r1))\ f) * \texttt{R}\ 2\ r2 * \texttt{R}\ 1\ r1)\}\end{aligned} \tag{4.2}$$

where p in Figure 4.2 is instantiated to $0x20$: the address where the instruction code is stored. Similarly, the Hoare rule for the second instruction of blk4 can be developed:

$$\begin{aligned}&\texttt{SPEC ARM\_INS}\ \{(0x24, \texttt{R}\ 1\ r1 * \texttt{S}\ \texttt{sZ}\ z\}\\ &\qquad (0x24, \texttt{0xE3310000})\ //\ \text{teq R1, \#0x0}\\ &\qquad \{(0x28, \texttt{R}\ 1\ r1 * \texttt{S}\ \texttt{sZ}\ (r1 = 0))\}.\end{aligned} \tag{4.3}$$

Notice that the value of the sZ flag is set to $(r1 = 0)$ in the postcondition. For clarity, I have omitted the assertions of other status flags whose values are also symbolic expressions. The judgments of the last instruction of the block are given before in Judgment (4.1).

With the Hoare judgments for individual instructions, we can compose them together to form Hoare judgments covering the block. Let us start with the first two instructions. The first step is to match state assertions used by the two judgments and use the Frame rule to add the assertions that are not used by a judgment to that judgment. In this process, the free variables of the second judgment are instantiated to the corresponding values in the postcondition of the first judgment. For example, assertion (MEMORY dom $f * \texttt{R}\ 2\ r2$) is added to Judgment (4.3), and the assertions of status flags are added to Judgment (4.2). The f symbolic variable in the second judgment is instantiated to $((r2 \mapsto (\texttt{w2w}\ r1))\ f)$, because f is free syntactically, and semantically the second judgment starts with the ending state of the first judgment.

The second step is to apply the Union rule to merge the two rules together.

The third step is using the Discharge rule to remove the intermediate entry in the postcondition, which is the entry with label $0x24$ in this case.

The last step is to apply the LPExt and Strengthen rules to remove the intermediate entry from the precondition to get a Hoare judgment for the two instructions.

Since this is a very straightforward process, I have omitted the intermediate results. The final judgment is:

$$
\begin{array}{l}
\texttt{SPEC ARM\_INS} \\
\quad \{(0x20, \texttt{MEMORY}\ dom\ f * \langle r2 \in dom \rangle * \texttt{R}\ 2\ r2 * \texttt{R}\ 1\ r1 * \texttt{S}\ \texttt{sZ}\ z)\} \\
\quad (0x20, \texttt{0xE5C21000})\ //\ \text{strb R1, [R2]} \\
\quad (0x24, \texttt{0xE3310000})\ //\ \text{teq R1, \#0x0} \\
\quad \{(0x28, \texttt{MEMORY}\ dom\ ((r2 \mapsto (\texttt{w2w}\ r1))\ f) * \texttt{R}\ 2\ r2 * \texttt{R}\ 1\ r1 * \texttt{S}\ \texttt{sZ}\ (r1 = 0))\}
\end{array}
$$

By repeating the same procedure with this judgment and the two judgments of the last instruction in Judgment (4.1), the judgments of blk4 are developed, and the final results are:

$$
\begin{array}{l}
\texttt{SPEC ARM\_INS} \\
\quad \{(0x20, \texttt{MEMORY}\ dom\ f * \langle r2 \in dom \rangle * \langle r1 \neq 0 \rangle * \texttt{S}\ \texttt{sZ}\ z * a1)\} \\
\quad \text{blk4} \\
\quad \{(0x14, \texttt{MEMORY}\ dom\ ((r2 \mapsto (\texttt{w2w}\ r1))\ f) * \texttt{S}\ \texttt{sZ}\ (r1 = 0) * a1)\} \\
\quad a1 = \texttt{R}\ 2\ r2 * \texttt{R}\ 1\ r1
\end{array}
\tag{4.4}
$$

$$
\begin{array}{l}
\texttt{SPEC ARM\_INS} \\
\quad \{(0x20, \texttt{MEMORY}\ dom\ f * \langle r2 \in dom \rangle * \langle r1 = 0 \rangle * \texttt{S}\ \texttt{sZ}\ z * a1)\} \\
\quad \text{blk4} \\
\quad \{(0x2C, \texttt{MEMORY}\ dom\ ((r2 \mapsto (\texttt{w2w}\ r1))\ f) * \texttt{S}\ \texttt{sZ}\ (r1 = 0) * a1)\}
\end{array}
\tag{4.5}
$$

It is noteworthy that the composition process is mechanical and does not require smart rule selection, so that it can be automated by metalanguage programming: the SML programming environment of the HOL theorem prover.

4.5.2.1 Pushing up Pure Assertions

After composition, the branch condition of an instruction is "pushed up" to the precondition of the judgment of the code block that contains the instruction, becoming

the block's precondition, such as the branch conditions $(r1 \neq 0)$ and $(r1 = 0)$ in the above example. If we merge Judgments 4.4 and 4.5 together by using the LPMerge rules discussed in Section 4.5.1, the two branch conditions become tautologous $\langle (r1 \neq 0) \vee (r1 = 0) \rangle$ and can be removed from the precondition of the merged judgment. The merged judgment of blk4 is shown below:

$$
\begin{array}{l}
\texttt{SPEC ARM\_INS} \\
\quad \{(0x20, \texttt{MEMORY}\ dom\ f * \langle r2 \in dom \rangle * \texttt{S}\ \texttt{sZ}\ z * a1)\} \\
\quad\ \text{blk4} \\
\quad \{(0x14, \texttt{MEMORY}\ dom\ ((r2 \mapsto (\texttt{w2w}\ r1))\ f) * \texttt{S}\ \texttt{sZ}\ (r1 = 0) * a1), \\
\quad\ (0x2C, \texttt{MEMORY}\ dom\ ((r2 \mapsto (\texttt{w2w}\ r1))\ f) * \texttt{S}\ \texttt{sZ}\ (r1 = 0) * a1)\}
\end{array}
\tag{4.6}
$$

I want to emphasize that a safety assertion to be discussed later when proving safety properties exhibits a similar behavior to the branch condition, namely, it is pushed up from the precondition of an instruction to the precondition of the code block containing the instruction, if it cannot be discharged or proven inside the code block. What is different, though, is that a safety assertion may not have two opposite cases to form a tautology.

4.5.3 Well-Formed Hoare Judgment

In order to model a code block which has only one entry address, I defined a *well-formed* Hoare judgment as a single-entry multiple-exit Hoare judgment by imposing two constraints: (1) there is only one entry address for the code; (2) the label of a label predicate in the precondition must be the entry address. Formally, it is

$$
\begin{array}{l}
\texttt{WF\_SPEC}\ ir\ P\ C\ Q\ \textbf{iff} \\
\qquad (\texttt{SPEC}\ ir\ \{P\}\ C\ \{Q\}) \wedge (\forall (l, p) \in P.\ \ l = \texttt{L}(C))
\end{array}
$$

where $\texttt{L}(C)$ is an auxiliary function that returns the entry address of a code block C, defined as $\texttt{L}(C) = \texttt{min}(\texttt{image fst}\ C)$, where a code block is represented as a set of labeled instructions, and the entry instruction has the lowest address. The `image` function returns the range of a domain: $\texttt{image}\ f\ s = \{f\ x \mid x \in s\}$; `fst` returns the first element of a tuple.

4.6 Hierarchical Function Judgment

The central structure of $\mathcal{L}_{fn}$ is a recursive function judgment. Informally, a function consists of code blocks and function calls. Code blocks are specified by the well-formed Hoare judgment described above. For function calls, I abstract a callee as a well-formed node, which behaves like a well-formed Hoare judgment in the caller. It has a single-entry precondition, abstract code and a multiple-exit postcondition. Now a function only has Hoare judgments, and the relationship among these Hoare judgments is specified as the following: for every judgment, the postcondition of its predecessors implies its precondition. If the function is called by another function, then the former can be abstracted again to act as a single Hoare judgment in the reasoning process about its caller. This recursive process forms a hierarchy of function judgments until reaching the entry or top-level function of a program.

4.6.1 Formal Definitions

This subsection presents the series of concepts introduced above formally.

4.6.1.1 Implication

The implication idea mentioned above originates from Floyd's inductive assertion [33], and I define it formally in order to assign meanings to functions:

$$Q \overset{P}{\Rightarrow} R \textbf{ iff } \forall (l,p) \in R.\ \forall (k,q) \in Q. \\ (k = l) \Rightarrow \left(\{(k,q)\} \overset{lp}{\Rightarrow} \{(l,p)\} \right).$$

It states that a set of label predicates Q implies another set of label predicates R (at the function level) if and only if for every label predicate lp in R, if a label predicate kq in Q has the same label with lp, then the singleton set of kq should imply the singleton set of lp.

4.6.1.2 Function Judgment

I define the hierarchical function judgment formally in Figure 4.4. There are two central definitions: Figure 4.4(a) shows the judgment of a function, and Figure 4.4(b) defines the concept of a well-formed node. In $\mathcal{L}_{fn}$, a function is a set of nodes with

$$\begin{array}{l}\texttt{FUN\_SPEC}\ wf\ ir\ prog\ entry\ init\ exits\ predecessor\ bspec\ kspec\ \textbf{iff}\\ \quad (\{(entry, init)\} \overset{lp}{\Rightarrow} (bspec\ (\texttt{bbl}\ entry)))\wedge\\ \quad (\forall (e, q) \in exits.\ (kspec\ e) \overset{lp}{\Rightarrow} q)\wedge\\ \quad (\forall node \in prog.\\ \qquad (wf\ ir\ (bspec\ node)\ node\ (kspec\ node))\wedge\\ \qquad (\forall pre \in (predecessor\ node).\ (kspec\ pre) \overset{P}{\Rightarrow} (bspec\ node)))\end{array}$$

(a) Function judgment

$$\frac{\texttt{WF\_SPEC}\ ir\ \{(l,p)\}\ C\ \{Q\}}{\texttt{WF\_NODE}\ ir\ \{(l,p)\}\ (\texttt{bbl}\ l)\ \{Q\}}\ \text{Base}$$

$$\frac{\texttt{FUN\_SPEC WF\_NODE}\ ir\ prog\ entry\ init\ exits\ predecessor\ bspec\ kspec}{\texttt{WF\_NODE}\ ir\ \{(entry, init)\}\ (\texttt{fun}\ entry)\ (\bigcup(\texttt{image snd}\ exits))}\ \text{Induction}$$

(b) Well-formed node

$$\begin{array}{l}\texttt{PROG\_SPEC}\ ir\ prog\ entryAddr\ predecessor\ bspec\ \textbf{iff}\\ \quad \exists kspec\ exits.\ \texttt{FUN\_SPEC WF\_NODE}\ ir\ prog\ entryAddr\ (\lambda s.\ \texttt{T})\\ \qquad\qquad\qquad\qquad\qquad\qquad exits\ predecessor\ bspec\ kspec\end{array}$$

(c) Program judgment as the judgment of the top-level function

Figure 4.4: Hierarchical function judgment

certain constraints, and the constraints specify the entry and exit conditions as well as the relationship among the nodes. Roughly speaking, the first two lines of the definition in Figure 4.4(a) restrict the entry and exit conditions with respect to the specifications of a function. The last three lines constrain the nodes of a function: the second to the last line requires that every node of the function is well-formed, whose definition will come later, and the last line specifies the relationship among the nodes.

I now explain the parameters of the function judgment in Figure 4.4(a). The first parameter of the definition, *wf*, is a well-formed node relation, and it refers to Hoare judgments of code blocks or the Hoare abstractions of function calls. Figure 4.4(b) defines such a relation. The second parameter *ir* is an instruction semantics, and

prog is the set of nodes of a function. The next parameter *entry* is the entry address of the function, and *init* is the initial condition of the function. The *exits* parameter is a set of pairs of an exit node and its associated exit condition. The *predecessor* parameter models the CFG policy at the node level within a function: given a node, it returns the set of predecessor nodes. The last two parameters *bspec* and *kspec* are specifications for all nodes of the function; the former is a mapping from nodes to their preconditions, and the latter is a mapping from nodes to their postconditions. The relationship among nodes specified in the last line of the definition states that if a node is a predecessor of another node, then the postcondition of the predecessor implies the precondition of the node. The constraints of the entry condition of a function specify that the initial condition of the function subsumes the *bspec* at the entry node, and the constraints of the exit condition stipulate that for every exit node, its *kspec* subsumes the exit condition associated with that node. In a simple case, $\{(entry, init)\}$ is (*bspec* (`bbl` *entry*)), and (*kspec* e) is q.

4.6.1.3 Abstract Code

The `bbl` function is one of the two constructors for the data type `fun_node`, which represents abstracted code, i.e., a code block or a function by its entry label:

`bbl,fun:` word32 $\rightarrow$ `fun_node`.

I use two constructors for human readability purposes, indicating that a node is a code block or a function abstraction; from the perspective of a type system, one constructor is enough.

4.6.1.4 Well-Formed Node

The concept of a well-formed node is central to the hierarchy of function judgments, which is formally defined in Figure 4.4(b) with the inductive relation definition of the metalogic. The Base rule states that the well-formed Hoare judgment of a code block is a well-formed node. The Induction rule states that a function judgment is also a well-formed node, given that the nodes of the function are already well-formed. The precondition of this well-formed node is the initial condition of the function, and

the postcondition is the union of all exit conditions of the function, represented by operator $\bigcup$. The `snd` function returns the second element of a tuple. In the call graph of a program, the leaf functions, which do not have a callee, only have the `bbl` nodes created from applying the Base rule to its code blocks; other functions have both `bbl` nodes and `fun` nodes, and the latter is generated by applying the Induction rule to judgments of callees.

In the metalogic, defining the Induction rule requires monotonicity of the antecedent with respect to the relation parameter [68], namely, one needs to show `FUN_SPEC` is monotonic with respect to its first parameter *wf*. Specifically, I proved the following lemma by using the definitions of `FUN_SPEC`, $\overset{lp}{\Rightarrow}$ and $\overset{P}{\Rightarrow}$:

$$(\forall ir\ P\ C\ Q.\ wf1\ ir\ P\ C\ Q \Rightarrow wf2\ ir\ P\ C\ Q) \Rightarrow$$
$$(\texttt{FUN\_SPEC}\ wf1\ ir\ prog\ entry\ init\ exits\ pred\ b\ k \Rightarrow$$
$$\texttt{FUN\_SPEC}\ wf2\ ir\ prog\ entry\ init\ exits\ pred\ b\ k).$$

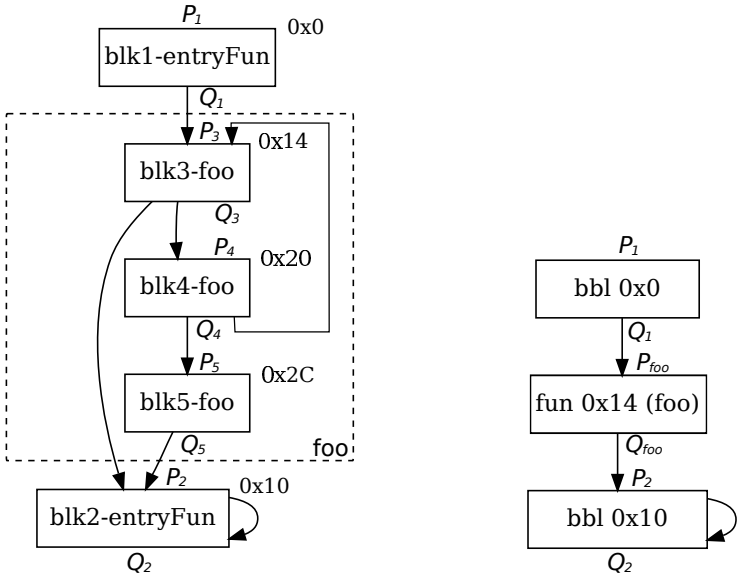
With the definition shown in Figure 4.4, a function judgment is `FUN_SPEC WF_NODE` in $\mathcal{L}_{fn}$. Although the definitions of `WF_NODE` are written in the natural deduction-style without quantifying any parameters, all the parameters of the two antecedents are universally quantified in the actual HOL logic, showing the nature of higher-orderness. Particularly, the definition of `FUN_SPEC` takes higher-order parameters, such as *wf* which may be instantiated with `WF_NODE`.

4.6.1.5 Program Judgment

Based on the above definitions, a program judgment simply becomes the judgment of the top-level or entry function. Its definition is given in Figure 4.4(c), where the initial condition is a true state predicate ($\lambda s.$ `T`).

4.6.2 An Example

I use the code example shown in Figure 1.3(a) to illustrate how the function judgment works in reasoning about a program. Suppose that we have developed the Hoare judgments of code blocks of the program. Denote the pre- and postconditions of block blk$i$ as P_i and Q_i, respectively. Figure 4.5(a) marks them in the code block level



(a) Code block judgments (b) `foo` abstraction

Figure 4.5: Function judgments

CFG of the program. Further suppose that we have derived the implication relation between adjacent pre- and postcondition pairs, namely the $\overset{P}{\Rightarrow}$ relation holds between pairs $(Q_1,\ P_3)$, $(Q_3,\ P_4)$, $(Q_3,\ P_2)$, (Q_4, P_3), $(Q_4,\ P_5)$, $(Q_5,\ P_2)$, and $(Q_2,\ P_2)$.

Under these assumptions, we can prove that Hoare judgments of code blocks are well-formed by the definition of well-formed Hoare judgment in Section 4.5.3, because each code block indeed has only one entry address. The proven well-formed Hoare judgments look like the following:

$$\texttt{WF\_SPEC ARM\_INS}\ \{P_i\}\ blk_i\ \{Q_i\} \qquad \text{for } 1 \leq i \leq 5.$$

Next, we apply the Base rule defined in Figure 4.4(b) to the above judgments to get well-formed nodes of code blocks:

$$\texttt{WF\_NODE ARM\_INS}\ \{P_i\}\ (\texttt{bbl}\ (\texttt{L}\ blk_i))\ \{Q_i\} \qquad \text{for } 1 \leq i \leq 5.$$

The judgment of function `foo` can be developed as follows. The most important parameters in the definition of function judgment in Figure 4.4(a) are the two mappings *bspec* and *kspec*. The former is the mapping from nodes to their precondition,

namely, it is the following, if I write the mapping as a list of update operations for readability purpose:

$$bspec\_foo = [(\texttt{bbl 0x14}) \mapsto P_3,\ (\texttt{bbl 0x20}) \mapsto P_4,\ (\texttt{bbl 0x2C}) \mapsto P_5]$$

where the entry address of each code block is explicitly written. Similarly, *kspec* is

$$kspec\_foo = [(\texttt{bbl 0x14}) \mapsto Q_3,\ (\texttt{bbl 0x20}) \mapsto Q_4,\ (\texttt{bbl 0x2C}) \mapsto Q_5].$$

The entry address of function `foo` is 0x14, and its initial condition is taken as the condition in P_3, i.e., $\{(entry\_foo, init\_foo)\} = P_3$ (remember that the precondition is a singleton of label predicate). The exit condition of the function is

$$exits\_foo = \{(\texttt{bbl 0x14},\ Q_3),\ (\texttt{bbl 0x2C},\ Q_5)\},$$

because both blk3, whose entry address is 0x14, and blk5, whose entry address is 0x2C, are exit blocks.

The CFG parameter is the node-level predecessor relation, and it can also be written as a mapping from a node to its predecessor nodes, namely,

$$\begin{aligned} predecessor\_foo = [&(\texttt{bbl 0x14}) \mapsto \{(\texttt{bbl 0x20})\}, \\ &(\texttt{bbl 0x20}) \mapsto \{(\texttt{bbl 0x14})\}, \\ &(\texttt{bbl 0x2C}) \mapsto \{(\texttt{bbl 0x20})\}] \end{aligned}$$

In reasoning about function `foo`, only its own code is considered, and its code is represented by a set of abstract nodes:

$$prog\_foo = \{(\texttt{bbl 0x14}), (\texttt{bbl 0x20}), (\texttt{bbl 0x2C})\}.$$

After constructing these terms, we can write down the judgment of function `foo`:

$$\begin{aligned} &\texttt{FUN\_SPEC WF\_NODE ARM\_INS } prog\_foo\ entry\_foo\ init\_foo\ exits\_foo \\ &\qquad\qquad predecessor\_foo\ bspec\_foo\ kspec\_foo \end{aligned}$$

It is easy to prove this judgment as a theorem, after expanding the definition of `FUN_SPEC`. For example, for node (`bbl 0x14`), its predecessor's postcondition is

kspec_foo (`bbl 0x20`) $= Q_4$, which implies ($\overset{P}{\Rightarrow}$) the precondition of the node, which is *bspec_foo* (`bbl 0x14`) $= P_3$, under the above assumptions.

We can now apply the Induction rule defined in Figure 4.4(b) to obtain the Hoare abstraction of function `foo`, after proving the above judgment.

$$\texttt{WF\_NODE ARM\_INS}\ \{(\mathit{entry\_foo},\ \mathit{init\_foo})\}\ (\texttt{fun 0x14})\ \left(\bigcup(\texttt{image snd}\ \mathit{exits\_foo})\right)$$
$$= \texttt{WF\_NODE ARM\_INS}\ P_3\ (\texttt{fun 0x14})\ (Q_3 \cup Q_5).$$

This abstraction gives function `foo` a logic syntax of Hoare triple, which can be in turn used just like a well-formed Hoare judgment of a node in reasoning about its caller `entryFun`—the top-level function of the program. Let $P_{foo} = P_3$ and $Q_{foo} = (Q_3 \cup Q_5)$. Then, we get the nodes and their pre- and postconditions of function `entryFun` shown in Figure 4.5(b). Repeating the same procedural for developing the function judgment of `foo`, we can develop the judgment of `entryFun` as:

$$\texttt{FUN\_SPEC WF\_NODE ARM\_INS}\ \mathit{prog\_f}\ \texttt{0x0}\ \mathit{init\_f}\ \mathit{exits\_f}$$
$$\mathit{pred\_f}\ \mathit{bspec\_f}\ \mathit{kspec\_f}$$

where

$$\mathit{prog\_f} = \{(\texttt{bbl 0x0}),\ (\texttt{fun 0x14}),\ (\texttt{bbl 0x10})\}$$
$$\{(\texttt{0x0},\ \mathit{init\_f})\} = P_1$$
$$\mathit{exits\_f} = \{(\texttt{bbl 0x10},\ Q_2)\}$$
$$\mathit{pred\_f} = [(\texttt{bbl 0x0}) \mapsto \{\},$$
$$(\texttt{fun 0x14}) \mapsto \{(\texttt{bbl 0x0})\},$$
$$(\texttt{bbl 0x10}) \mapsto \{(\texttt{fun 0x14}),\ (\texttt{bbl 0x10})\}]$$
$$\mathit{bspec\_f} = [(\texttt{bbl 0x0}) \mapsto P_1,\ (\texttt{fun 0x14}) \mapsto P_{foo},\ (\texttt{bbl 0x10}) \mapsto P_2]$$
$$\mathit{kspec\_f} = [(\texttt{bbl 0x0}) \mapsto Q_1,\ (\texttt{fun 0x14}) \mapsto Q_{foo},\ (\texttt{bbl 0x10}) \mapsto Q_2].$$

Likewise, this judgment can be proven by expanding the definition of `FUN_SPEC` under the same assumptions.

The program judgment naturally follows according to its definition in Figure 4.4(c), once the above top-level function judgment is proven:

$$\texttt{PROG\_SPEC ARM\_INS}\ prog\_f\ \texttt{0x0}\ pred\_f\ bspec\_f. \tag{4.7}$$

4.6.3 Discussion

The function presented here is a concept in logic, although it mirrors a function at the object-code level. A function may or may not coincide with a function in the source code language. Similarly, it may or may not correspond to a function at the object-code level, although in practice I simply abstract an object-code level function into a logic function due to its convenience. The object-code level functions may be constructed by a binary analysis/rewriting tool. Such a tool can decompile an executable, construct a conservative CFG, and build functions by using call/return conventions, among other functionalities [50, 104].

4.6.3.1 Unstructured Control Flow Transfers

Although some inference rules are proven for the Hoare logic used in $\mathcal{L}_{fn}$ in Section 4.5, they are only used to compose judgments for code blocks, because the sequential structure of code blocks is simple enough that the composition process can be automated. Unstructured control flow transfers are reasoned about in the hierarchical function judgment, and this reasoning process, as illustrated in the previous subsection, does not require any rules. It requires the establishment of the implication relation, $\overset{P}{\Rightarrow}$, between two adjacent Hoare judgments of function nodes and the Hoare abstraction of functions.

4.6.3.2 Partial and Total Correctness

The boundary of partial and total correctness in a traditional Hoare logic [25] disappears in $\mathcal{L}_{fn}$. There is no requirement for a termination proof. Terminating or nonterminating code can be reasoned about in the same way without worrying about termination at all. All that is necessary is to find the implication relation. This has real-world applications, where many control loops in embedded systems are purposely written to execute forever.

4.6.3.3 Proving Safety Properties

$\mathcal{L}_{fn}$ does not directly encode a safety property itself. However, it supports making a safety assertion at every state with the use of the parametric instruction semantics throughout the logic. For a certain safety property of interest, a user can define a customized instruction semantics which asserts the safety property, and then use it to instantiate the semantic parameter. This means that the safety property is asserted at every instruction of the program and hence for all states of a program. This is to be described in detail when ARMor's verification framework is presented in Chapter 6.

4.6.3.4 Hierarchical Reasoning

$\mathcal{L}_{fn}$ naturally divides a program into two types of proof units. The first kind of proof units is a node with its predecessors, and it enables reasoning about a function locally. The second kind is a function, and it reasons about the whole program hierarchically. This feature is particularly useful in proving shallow safety properties, because it allows to automate the entire verification process by leveraging whole-program interprocedural abstract interpretation to find the specifications of functions. Again, this is to be discussed in Chapter 6.

4.7 Soundness

The soundness proof for $\mathcal{L}_{fn}$ states that a program never gets stuck under a given semantics throughout its execution. An informal argument is that when control reaches the end of a code block, it resumes on one of its successor blocks (including jumping to the entry block of another function) because of the implication relation. Formally, a function specification `FUN_SPEC` may be derived if and only if: starting from its initial state s, if the execution reaches the label of a code block, $\mathtt{L}(n)$, then the precondition defined by *bspec* on the block is ensured to be true. The corresponding theorem is

$$\forall ir\ s\ sq\ k\ n.$$
$$(\mathtt{seq}\ ir\ C\ sq\ s) \wedge (bspec\ (\mathtt{bbl\ entry})\ s) \wedge (\mathtt{LABEL\_IN}\ (\mathtt{L}(n))\ (sq\ k))$$
$$\Rightarrow (bspec\ n)\ (sq\ k)$$

where LABEL_IN specifies that a state has a label, or the control reaches to the state: LABEL_IN l t **iff** $\exists p.\ ((\texttt{LP2SP}(l, p))\ t)$, and C is the set of code blocks of the function and all its callees.

4.8 Discussion of Related Work

Hoare logic and its variations have been well studied, and Section 2.2.1 introduced their development. $\mathcal{L}_{fn}$ is designed to address the issues inherent in a Hoare logic, so that verification of safety properties of low-level or machine-code programs can be effectively automated in a mechanized theorem prover. However, I did not discard Hoare logic completely, because it is useful in certain situations. As described in this chapter, the Hoare logic is used to reason about code blocks in $\mathcal{L}_{fn}$, because code blocks only have sequential control flow transfers, and this simple structure can be reasoned about very efficiently by Hoare logic. At the program level, Hoare logic describes a program by a Hoare triple, while $\mathcal{L}_{fn}$ describes a program by the top-level function judgment. Concrete examples of program judgments are Theorems (1.1) and (4.7).

Section 2.2.2 introduced CAP family of program logics. At a high level, $\mathcal{L}_{fn}$ is similar to CAP in some sense: neither has complicated rule systems, and both formalize Floyd's inductive assertion idea. However, they also have important differences. CAP verifies a program specification one instruction at a time: it needs to write down intermediate assertions for every instruction interactively, while $\mathcal{L}_{fn}$ uses a Hoare logic to generate specifications for code blocks automatically. For shallow safety properties such as the isolation properties verified by ARMor, the structure of $\mathcal{L}_{fn}$ enables automatic generation of program specifications. In contrast, CAP related work does not discuss how program specifications are effectively developed.

The development of $\mathcal{L}_{fn}$ is influenced by existing work. For example, Appel et al. defined a safety property into a type system such that the type-safety of a program implies the safety property of interest [5,6]. In supporting proof reuse, Myreen et al. used a parametric instruction semantics in their architecture-independent Hoare logic and instantiated the parameter with concrete instruction semantics of ARM, x86, and

PowerPC [77]. $\mathcal{L}_{fn}$ uses a combination of these two: parameterizing the instruction semantics as the bottom layer and defining a safety property in the semantics. A concrete example is to be given in Section 6.2.

The top layer of $\mathcal{L}_{fn}$ formalizes Floyd's inductive assertion. Floyd proposed inductive assertion to verify programs in 1960s [33], and this idea has had extensive and profound influence in verification communities. For example, VCG and PCC both use it to discharge the conditions at certain program locations in program verification [48, 54, 79, 109]. More recently, CAP family of techniques formalizes it in a mechanized proof assistant and shows that it is possible to not use a Hoare-style reasoning system in theorem proving [83, 113]. This idea has greater flexibility than Hoare-style logics, such as reasoning about arbitrary control flow transfers and no need for termination proof.

The function concept in $\mathcal{L}_{fn}$ is inspired by function summary, which has long been used in interprocedural static analyses [26, 51, 112]. Traditional Hoare logics develop function call rules to compose the judgment of a callee into the judgment of a caller [75, 93]. In contrast, $\mathcal{L}_{fn}$ creates an abstract node for a callee in the caller, and the semantics of this node is specified by a Hoare triple, which behaves similarly to a Hoare judgment of a regular code block. There is no composition between the judgments of a caller and a callee.

CHAPTER 5

FORMALIZATION OF SAFETY PROPERTIES

ARMor verifies the memory safety and control flow integrity properties introduced in Section 1.3. In order to carry out formal verification, these properties need to be defined mathematically. This chapter gives formal definitions of the two properties.

5.1 Safety Properties Revisited

It is helpful to examine the characteristics of a safety property first. The memory safety stated in Section 1.3 can be rephrased as:

for every instruction of a binary program, the set of addresses it writes to is a subset of some given memory set.

This rephrasing reveals a fixed pattern about this type of safety property, and it has two characteristics: (1) *Every instruction* of a program should have this property. Because machine instructions are the minimum execution units observable from the perspective of a program, this means that *every state* of the program should have the property. (2) The property can be expressed as a *predicate* that involves some *attributes* of an instruction. In this case, the attributes are the set of addresses an instruction writes to, and the predicate is that this set is a subset of a given set. The two characteristics ensure that memory safety can be defined for every state of a program.

Control flow integrity can be rephrased as: *for every instruction of a binary program, the PC value it assigns to is one of its successor addresses specified in the given CFG policy.* The corresponding attribute is the PC value after an instruction executes, and the corresponding predicate can be defined as a set membership test on the attribute, if the successor addresses are represented by a set.

The same pattern may be applied to other safety properties, such as memory read safety: for every instruction of a program, the memory addresses it loads data from is a subset of some given memory address set. However, for purposes of this dissertation, I omit discussion of other safety properties.

5.2 Refinement of Memory Assertions

I augmented the existing ARM semantics introduced in Section 4.3 to make it fit for verifying isolation properties required by ARMor. The augmentation includes two major changes: refining memory assertions to reflect the introduction of the control stack described in Section 3.2.2, and formulating the isolation properties mathematically. The augmentation itself is proven as theorems in the HOL proof assistant.

The general form of theorems of the augmented semantics is

$$\begin{array}{l} \{\texttt{PC}\ l * \texttt{R}\ 8\ k * \langle l' \in \texttt{succ}(l)\rangle * \langle \mathit{MemorySafe}\rangle * \\ \qquad \texttt{MEMORY dm}\ \mathit{df} * \texttt{MEMORY cm}\ \mathit{cf} * \texttt{MEMORY pm}\ \mathit{pf} * p\} \\ (l,\ \mathit{ins}) \\ \{\texttt{PC}\ l' * \texttt{R}\ 8\ k' * \ \texttt{MEMORY dm}\ \mathit{df}' * \texttt{MEMORY cm}\ \mathit{cf}' * \texttt{MEMORY pm}\ \mathit{pf} * q\} \end{array} \tag{5.1}$$

where l is the value of the PC in the precondition, k is the value of the control stack pointer, p and q represent other assertions that are not explicitly written out, and corresponding values of machine resources in the postcondition are marked with a prime $'$.

The data memory of a program is divided into three parts and described by three separate assertions in the augmented theorems. The three parts are the writable data memory (WD), the control stack (CS), and the data pool (DP). WD includes the data stack, the global data section, and the I/O addresses as depicted in Figure 3.5. WD and CS form the given memory set mentioned in the memory safety policy described in Section 5.1 and is denoted as `mem` in Section 1.7. They are preallocated in a system configuration; the data pool is part of the code section of a program. Formally, they are modeled as three sets of addresses: `dm` represents WD, `cm` represents CS, and

pm represents DP. They must be disjoint in a system. Consequently, three separate assertions are used to describe their contents: MEMORY dm df asserts dm, MEMORY cm cf asserts cm, and MEMORY pm pf asserts pm. These memory assertions are explicitly written out to utilize a feature provided by separation logic [90], namely, memory addresses in dm, cm, and pm cannot appear in other machine resource assertions, and doing so will result in a false value. Notice that the data pool values stay the same in the pre- and postconditions as pf, indicating that the contents of the data pool cannot be changed.

Figure 5.1 shows a concrete example of the augmented theorems for instruction strb R1,[R2], whose original semantics is in Figure 4.2. In this example, only the value of the writable memory set dm is updated to $df' = ((r2 \mapsto (\texttt{w2w}\ r1))\ df)$, while the content of the control stack cm stays the same as in the precondition ($cf' = cf$).

5.3 Formal Definitions

Besides the memory assertions, each augmented theorem has two important pure assertions for the isolation properties. One is the assertion of control flow integrity:

$$l' \in \texttt{succ}(l)$$

where l and l' are the PC values before and after the execution of the instruction, and succ(l) returns the set of successor addresses for a given instruction address. Note that succ models the given CFG of a program by a function from address to address set, whose type is word32 $\rightarrow$ word32 set. This assertion formalizes the predicate of the control flow integrity discussed in Section 5.1.

$$\begin{aligned}
&\{\texttt{PC}\ p * \texttt{R}\ 8\ k * \langle (p+4) \in \texttt{succ}(p)\rangle * \langle \{r2\} \subseteq \texttt{dm}\rangle * \\
&\qquad \texttt{MEMORY dm}\ df * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf * \texttt{R}\ 2\ r2 * \texttt{R}\ 1\ r1\} \\
&\ (p, \texttt{0xE5C21000})\ //\ \texttt{strb R1,[R2]} \\
&\{\texttt{PC}\ (p+4) * \texttt{R}\ 8\ k\ * \texttt{MEMORY dm}\ ((r2 \mapsto (\texttt{w2w}\ r1))\ df) * \texttt{MEMORY cm}\ cf\ * \\
&\qquad \texttt{MEMORY pm}\ pf * \texttt{R}\ 2\ r2 * \texttt{R}\ 1\ r1\}
\end{aligned}$$

Figure 5.1: The augmented theorem of strb R1,[R2]

The other is the memory safety assertion denoted by $\langle MemorySafe \rangle$, whose complete form is

$$\begin{aligned} MemorySafe = \text{if } (\texttt{not } (\texttt{isStore } ins)) \text{ then } \texttt{true} \text{ else} \\ \texttt{ms}(ins) \subseteq (\text{if } k = k' \text{ then } \texttt{dm} \text{ else } \texttt{cm}) \end{aligned}$$

where $\texttt{ms}(ins)$ is the set of memory addresses that instruction ins writes to. The rationale of (if $k = k'$ then dm else cm) is that if the control stack pointer does not change during execution, then the instruction should write to dm; otherwise, it writes to cm. This ensures that changes to the control stack can only be done through its pointer. This assertion formalizes the predicate of the memory safety described in Section 5.1.

In the example of Figure 5.1, the assertion of control flow integrity is

$$(p + 4) \in \texttt{succ}(p), \tag{5.2}$$

and the assertion of memory safety is

$$\begin{aligned} memok = \text{if } \texttt{not } (\texttt{isStore 0xE5C21000}) \text{ then } \texttt{true} \text{ else} \\ \texttt{ms}(\texttt{0xE5C21000}) \subseteq (\text{if } k = k' \text{ then } \texttt{dm} \text{ else } \texttt{cm}), \end{aligned}$$

where *memok* denotes the specific instance of memory safety for the store instruction, which can be simplified to:

$$\begin{aligned} memok &= \texttt{ms}(\texttt{0xE5C21000}) \subseteq \texttt{dm} \\ &= \{r2\} \subseteq \texttt{dm} \end{aligned} \tag{5.3}$$

5.4 Proof Process

The augmented semantics is proven as theorems in the HOL system. The proof process involves two major logic operations as well as SML programming. The first logic operation is using the Frame rule of the existing Hoare logic to add missing memory assertions [75].

The second one is using the DISCH rule of the metalogic to introduce a boolean expression into the theorem obtained in the previous step and moving the expression

into the precondition of the theorem with the SPEC_MOVE_COND rule of the existing Hoare logic. These two rules are:

$$\frac{Assumptions \vdash t}{Assumptions - b \vdash b \Rightarrow t}\ \text{DISCH}$$

$$\frac{}{(b \Rightarrow \{P\}\ C\ \{Q\}) = \{P * \langle b \rangle\}\ C\ \{Q\}}\ \text{SPEC\_MOVE\_COND.}$$

In the next chapter, I will describe how the formal definitions of the two isolation properties defined here are integrated into ARMor's verification framework, and verified.

CHAPTER 6

AUTOMATED VERIFICATION FRAMEWORK

The framework of ARMor's formal verification is developed based on the theory of $\mathcal{L}_{fn}$ presented in Chapter 4. It aims to address two issues in verifying safety properties of machine-code programs in a higher-order logic proof assistant: asserting the safety properties for all states of a program and automating the entire verification process including specification generation.

The middle layer of $\mathcal{L}_{fn}$ is a Hoare logic, and safety properties such as the isolation properties introduced before are not directly amenable to Hoare-style reasoning, because a Hoare judgment only has state assertions before and after the execution of code without mentioning intermediate states, which is not sufficient in verifying safety properties. Safety properties require that every state of a program must not violate a given policy, which demands that every state of the program must have assertions of these properties. For example, if used directly to reason about the control flow integrity property discussed in Section 1.3, the Hoare judgment for a snippet of code `add R0,R0,#1; mov PC,R14` is

$$\begin{aligned} &\texttt{SPEC ORG\_INS}\ \{(p, \texttt{R}\ 0\ r0 * \texttt{R}\ 14\ r14)\} \\ &\qquad (p,\ \texttt{add R0,R0,\#1}) \\ &\qquad (p+4,\ \texttt{mov PC,R14}) \\ &\qquad \{(r14, \texttt{R}\ 0\ (r0+1) * \texttt{R}\ 14\ r14)\}. \end{aligned} \tag{6.1}$$

It is not clear if the control flow integrity holds, no matter what value $r14$ takes. Even worse, a piece of code may jump to illegal addresses and later jump back to legal addresses, and it is impossible to use the composed Hoare judgment to argue that the code violates the control flow integrity; conversely, it is also impossible to

argue the opposite: the code does not violate the control flow integrity. Either way, the Hoare judgment is insufficient.

The second issue is largely ignored by the formal verification community. Verification automation is traditionally thought of as a process of checking code implementation against its specification, and the specification is manually written [55]. For example, some work simply assumes the existence of correct specifications and does not discuss how they are developed [83, 113]. I take a different position on this issue: the entire verification process including both *developing* and *verifying* the specification should be automated for checking shallow safety properties of machine-code programs, because it is error-prone and inefficient to write specifications manually for such low-level programs, if it is possible.

6.1 Overview

The bottom layer and the higher-order semantic parameter of $\mathcal{L}_{fn}$ provide a mechanism that can be used to assert safety properties at every state of a program explicitly. The basic idea is asserting the safety properties in an existing semantics for every instruction, resulting in a customized semantic relation, which is used to instantiate the semantic parameter. The customized semantic relation is created by using the inductive relation definition introduced in Section 1.4.3, which guarantees that every state in the customized semantics has assertions of the safety properties. This in turn ensures that the instantiation of the semantic parameter by the customized semantics brings about the desired result: reasoning about the program is strictly based on the semantics that has asserted the safety properties at every state of the program.

Assertions of the safety properties in the existing semantics are carried over by the instantiation to the reasoning process of a program in $\mathcal{L}_{fn}$. These assertions must be discharged inside the program in order to meet the requirement of the program judgment, which has the weakest initial condition of true, i.e., $(\lambda s.\ \mathtt{T})$ as discussed in Section 4.6.1.5. In $\mathcal{L}_{fn}$, these assertions are pure in the terminology of separation logic. In order to emphasize their importance in verifying safety properties, I call

them *safety* assertions in this dissertation. In this framework, safety assertions are propagated and discharged by a whole-program interprocedural abstract interpretation, which automatically discovers the function specifications required by $\mathcal{L}_{fn}$ to prove a program judgment.

At a high level, the framework operates in the following steps. The first step is formulating safety properties as safety assertions in a customized instruction semantics. This step is completed manually, because different safety properties have different mathematical formulations. However, once the formulation is defined, it is reused to verify the same properties of different programs. For example, ARMor only formulates the two isolation properties once in Section 5.3, but automatically verifies different sandboxed ARM executables.

The second step is instantiating the semantic parameter of $\mathcal{L}_{fn}$, ir, with the customized instruction semantics. This instantiation has a very important implication in verifying the safeties properties: it guarantees that reasoning about a program is based on the customized semantics that asserts the safety properties at every state of the program.

The third step is discharging the safety assertions inside code blocks by using Hoare reasoning provided by the middle layer of $\mathcal{L}_{fn}$. Some safety assertions may be discharged, and some may not. The undischarged safety assertions comprise part of the precondition of code block judgments and are called safety assertions of code blocks.

The fourth step is discharging the safety assertions of code blocks globally through a whole-program interprocedural abstract interpretation. The result of the abstract interpretation describes where the safety assertions of code blocks are discharged along which call paths.

The fifth step is constructing function specifications based on the result of the abstract interpretation and proving function judgments. It conducts necessary transformations on code block judgments such that the required implication relation, $\overset{P}{\Rightarrow}$, defined in Section 4.6, holds between any two adjacent code blocks. Function judgments are proven, and the judgment of the top-level function is the judgment of

the program.

This chapter illustrates these steps except the third one by describing ARMor's verification in detail; the Hoare reasoning process used in the third step has been described in Section 4.5.2.

6.2 Customized Instruction Semantics

The purpose of developing a customized instruction semantics is to integrate the safety properties formalized in Chapter 5 into this verification framework.

The augmented semantics shown in Theorem (5.1) has the safety assertions needed to verify the isolation properties, but it cannot be directly used in $\mathcal{L}_{fn}$, because it does not have the label predicate syntax required by the latter. In order to utilize the augmented semantics, I use the inductive relation definition of the metalogic described in Section 1.4.3 to define a customized semantic relation, `SAFE_INS`, whose instances can only be created by the single rule, SafeIns, which is defined in Figure 6.1. The antecedent of the rule is Theorem (5.1), and the conclusion is the new semantic relation.

The rule plays two very important roles in verifying the isolation properties. First, it inherits the refined data memory assertions and isolation property assertions described before. This means that every state of a program in $\mathcal{L}_{fn}$ has assertions of the isolation properties, if a judgment has the semantic parameter `SAFE_INS`. Second,

$$\frac{\begin{array}{l}\{\texttt{PC}\ l * \texttt{R}\ 8\ k * \langle l' \in \texttt{succ}(l)\rangle * \langle \mathit{MemorySafe}\rangle * \\ \qquad \texttt{MEMORY dm}\ \mathit{df} * \texttt{MEMORY cm}\ \mathit{cf} * \texttt{MEMORY pm}\ \mathit{pf} * p\} \\ \quad (l,\ \mathit{ins}) \\ \{\texttt{PC}\ l' * \texttt{R}\ 8\ k' * \ \texttt{MEMORY dm}\ \mathit{df}' * \texttt{MEMORY cm}\ \mathit{cf}' * \texttt{MEMORY pm}\ \mathit{pf} * q\}\end{array}}{\begin{array}{l}\texttt{SAFE\_INS}\ (l,\ \texttt{R}\ 8\ k\ * \langle l' \in \texttt{succ}(l)\rangle * \langle \mathit{MemorySafe}\rangle * \\ \qquad \texttt{MEMORY dm}\ \mathit{df} * \texttt{MEMORY cm}\ \mathit{cf} * \texttt{MEMORY pm}\ \mathit{pf} * p) \\ \quad (l,\ \mathit{ins}) \\ \quad (l',\ \texttt{R}\ 8\ k' * \ \texttt{MEMORY dm}\ \mathit{df}' * \texttt{MEMORY cm}\ \mathit{cf}' * \texttt{MEMORY pm}\ \mathit{pf} * q)\end{array}}\ \text{SafeIns}$$

Figure 6.1: Safe instruction rule

the new semantic relation uses the label predicate syntax, so that the label component of a state predicate can be easily decomposed and matched by the metalogic, while the original separating conjunction operator ($*$) does not have this convenient reasoning power, because it is not a constructor.

Figure 6.2 shows an example of the customized semantics for instruction `strb R1,[R2]`, which is derived by applying the SafeIns rule to the augmented theorem in Figure 5.1. It is worth pointing out that every axiom describing a state transition in the existing semantics has a corresponding `SAFE_INS` relation instance proven in ARMor.

6.3 Instantiation with Customized Semantics

The second step of the framework is to instantiate the semantic parameter of $\mathcal{L}_{fn}$ with the customized instruction semantics. In ARMor's case, the instantiation results in different judgments for different code units, and they are given in Table 6.1.

The presence of `SAFE_INS` in the judgments indicates that the reasoning process of ARMor is strictly based on the customized safe instruction semantics in which

$$\begin{aligned}
&\texttt{SAFE_INS}\ (p, \texttt{R}\ 8\ k * \langle (p+4) \in \texttt{succ}(p) \rangle * \langle \{r2\} \subseteq \texttt{dm} \rangle * \\
&\qquad \texttt{MEMORY dm}\ df * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf * \texttt{R}\ 2\ r2 * \texttt{R}\ 1\ r1) \\
&\quad (p, \texttt{0xE5C21000})\ //\ \texttt{strb R1,[R2]} \\
&\quad (p+4, \texttt{R}\ 8\ k\ * \texttt{MEMORY dm}\ ((r2 \mapsto (\texttt{w2w}\ r1))\ df) * \texttt{MEMORY cm}\ cf\ * \\
&\qquad \texttt{MEMORY pm}\ pf * \texttt{R}\ 2\ r2 * \texttt{R}\ 1\ r1)
\end{aligned}$$

Figure 6.2: Customized semantics of `strb R1,[R2]`

Table 6.1: Instantiated judgments

Judgment	**Safe Version**
Code block judgment	`SPEC SAFE_INS`
Well-formed Hoare judgment	`WF_SPEC SAFE_INS`
Well-formed node	`WF_NODE SAFE_INS`
Function judgment	`FUN_SPEC WF_NODE SAFE_INS`
Program judgment	`PROG_SPEC SAFE_INS`

every instruction, and thus every state, has the assertions of the isolation properties. When written out in a mathematical formula, the result of the judgment `PROG_SPEC SAFE_INS` proven by ARMor's verification is equivalent to the following:

Theorem 1 $\forall i \in \mathit{prog}.\,(\mathit{ms}(i) \subseteq \mathit{mem}) \wedge (\mathit{pc\_after}(i) \in \mathit{succ}(\mathit{address\_of}(i)))$ *with respect to the initial state* $(\lambda s.\ T)$.

where `prog` represents a program, and `address_of` and `pc_after` return the values of PC before and after an instruction i executes, respectively, corresponding to l and l' in Theorem (5.1). It reads as that for every instruction of the program, the memory addresses it writes to are a subset of the given memory set, and the PC value after the instruction is in the given set of successor addresses.

6.4 Safety Assertion Analysis

One of the central tasks of ARMor's verification is to derive the two function specifications *bspec* and *kspec*, such that the required implication relation between adjacent code block pairs holds, namely, postconditions of the predecessors of a code block imply the precondition of that code block. The implication relation is formally specified as $(kspec\ pre) \stackrel{P}{\Rightarrow} (bspec\ node)$ in the definition of `FUN_SPEC` in Figure 4.4. The third step composes Hoare judgments of code blocks, and these judgments may have safety assertions that cannot be discharged by the Hoare reasoning process itself. These safety assertions present challenges to developing correct function specifications, because a safety assertion for a code block may require other code blocks to be enhanced with some forms of the safety assertion. An example can illustrate this issue best.

6.4.1 Challenges in Global Reasoning

The following is the resultant judgment for code block blk4 and blk5 of the example shown in Figure 1.3(a) after the first three steps in this framework. For clarity of presentation, I have used single variables with a prime such as c' to represent some uninteresting values of status flags in the postcondition; relevant values to this

illustration are written out explicitly such as the value for zero flag sZ. The values of status flags in the postcondition are set by the test equality instruction teq.

The Hoare judgments of blk4 are

$$\begin{array}{l}\texttt{SPEC SAFE\_INS}\ \{(0x20, \texttt{R}\ 8\ k\ *\texttt{REG}\ rf * \langle\{rf\ R2\} \subseteq \texttt{dm}\rangle\ * \langle rf\ R1 \neq 0x0\rangle * \\ \qquad\qquad \texttt{MEMORY dm}\ df * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf\ * \\ \qquad\qquad \texttt{S sC}\ c * \texttt{S sN}\ n * \texttt{S sV}\ v * \texttt{S sZ}\ z)\} \\ \qquad\qquad \text{blk4} \\ \{(0x14, \texttt{R}\ 8\ k\ *\texttt{REG}\ rf * \texttt{MEMORY dm}\ ((rf\ R2 \mapsto \texttt{w2w}(rf\ R1))\ df) * \\ \qquad\qquad \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf\ * \\ \qquad\qquad \texttt{S sC}\ c' * \texttt{S sN}\ n' * \texttt{S sV}\ v' * \texttt{S sZ}\ ((rf\ R1) = 0x0))\}\end{array} \tag{6.2}$$

and

$$\begin{array}{l}\texttt{SPEC SAFE\_INS}\ \{(0x20, \texttt{R}\ 8\ k\ *\texttt{REG}\ rf * \langle\{rf\ R2\} \subseteq \texttt{dm}\rangle\ * \langle rf\ R1 = 0x0\rangle * \\ \qquad\qquad \texttt{MEMORY dm}\ df * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf\ * \\ \qquad\qquad \texttt{S sC}\ c * \texttt{S sN}\ n * \texttt{S sV}\ v * \texttt{S sZ}\ z)\} \\ \qquad\qquad \text{blk4} \\ \{(0x2C, \texttt{R}\ 8\ k\ *\texttt{REG}\ rf * \texttt{MEMORY dm}\ ((rf\ R2 \mapsto \texttt{w2w}(rf\ R1))\ df) * \\ \qquad\qquad \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf\ * \\ \qquad\qquad \texttt{S sC}\ c' * \texttt{S sN}\ n' * \texttt{S sV}\ v' * \texttt{S sZ}\ ((rf\ R1) = 0x0))\}.\end{array} \tag{6.3}$$

The Hoare judgment of blk5 is

$$\begin{array}{l}\texttt{SPEC SAFE\_INS}\ \{(0x2C, \texttt{R}\ 8\ k\ *\texttt{REG}\ rf * \langle rf\ R14 \in \texttt{succ}(0x2C)\rangle\ * \\ \qquad\qquad \texttt{MEMORY dm}\ df * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf)\} \\ \qquad\qquad \text{blk5} \\ \{(rf\ R14, \texttt{R}\ 8\ k\ *\texttt{REG}\ rf\ * \\ \qquad\qquad \texttt{MEMORY dm}\ df * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf)\}.\end{array} \tag{6.4}$$

The symbolic labels or values of PC have been instantiated to the concrete addresses of corresponding instructions. Each judgment has undischarged safety asser-

tions: Judgments 6.2 and 6.3 have a memory safety assertion: $\langle\{rf\ R2\} \subseteq \texttt{dm}\rangle$, and Judgment 6.4 has a control flow integrity assertion: $\langle rf\ R14 \in \texttt{succ}(0x2C)\rangle$. The control flow integrity assertions of the instructions in blk4 have been simplified to `true`, namely, been discharged. Discharged is also the memory safety assertion of the instruction in blk5.

In order to prove the implication relation between blk4 and blk5, we need to show that the postcondition of Judgment 6.3 implies the precondition of Judgment 6.4, because they have the same label. However, this is not true for the current results. The only way to make it true is to strengthen the postcondition of Judgment 6.3 to include the term $\langle rf\ R14 \in \texttt{succ}(0x2C)\rangle$, so that the strengthened postcondition can imply the control flow integrity assertion of Judgment 6.4. However, there is not a strengthening rule for postconditions, only the Weaken rule as shown in Figure 4.3. An inspiring question is what changes can be made to Judgment 6.3, if we want its postcondition to imply the safety assertion term? A solution is to use the Frame rule to add the pure assertion to both the pre- and postcondition of Judgment 6.3. This results in the following judgment:

$$
\begin{aligned}
&\texttt{SPEC SAFE\_INS}\ \{(0x20, \texttt{R}\ 8\ k\ *\texttt{REG}\ rf * \langle\{rf\ R2\} \subseteq \texttt{dm}\rangle\ * \langle rf\ R1 = 0x0\rangle * \\
&\qquad \texttt{MEMORY dm}\ df * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf\ * \\
&\qquad \texttt{S sC}\ c * \texttt{S sN}\ n * \texttt{S sV}\ v * \texttt{S sZ}\ z\ * \\
&\qquad \langle rf\ R14 \in \texttt{succ}(0x2C)\rangle)\} \\
&\qquad \text{blk4} \qquad\qquad (6.5)\\
&\{(0x2C, \texttt{R}\ 8\ k\ *\texttt{REG}\ rf * \texttt{MEMORY dm}\ ((rf\ R2 \mapsto \texttt{w2w}(rf\ R1))\ df) * \\
&\qquad \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf\ * \\
&\qquad \texttt{S sC}\ c' * \texttt{S sN}\ n' * \texttt{S sV}\ v' * \texttt{S sZ}\ ((rf\ R1) = 0x0) * \\
&\qquad \langle rf\ R14 \in \texttt{succ}(0x2C)\rangle)\}.
\end{aligned}
$$

Now, we can prove that the postcondition of Judgment 6.5 implies the precondition of Judgment 6.4. Both Judgments 6.3 and 6.5 are valid Hoare judgments for blk4, but only Judgment 6.5 is the desired one, because it enables a successful proof.

However, a new but similar question arises for discharging the safety assertions of blk4, which had only one memory assertion before, but now have an additional control flow integrity assertion. Imagine a program whose safety assertions scatter in code blocks of different functions; the question becomes how to develop the desired judgments for each code block efficiently. The fundamental reason causing this issue is that Hoare reasoning is local and does not have the required global information.

In theory, we can repeat the above reasoning process for blk3 and blk4 to discharge the safety assertions of blk4; this chain of reasoning can continue until some point where the safety assertions can be discharged. For example, when $\langle rf\ R14 \in \texttt{succ}(0x2C)\rangle$ is framed to the judgment of blk3, its propagation stops, because the postcondition of blk1 can imply it. The call instruction `bl` at the end of blk1 places the return address $0x10$ into the link register R14, and this address value is indeed in the set returned from $\texttt{succ}(0x2C)$, which is part of the safety policy given in Figure 1.3(b).

Obviously, this process cannot be done efficiently by human effort for a machine-code program. Fortunately, it may be conducted automatically by leveraging the general framework of a whole-program interprocedural abstract interpretation. I will describe this proof analysis as much as possible in the terminologies of abstract interpretation with differences pointed out.

The central question that the analysis answers, as illustrated in the above example, is what new judgments should be developed for a code block in order to prove the implication relation between the judgments of the code block and the judgments of the successors of the code block. When the original judgments of code blocks, as the results of the Hoare reasoning, are viewed as nodes similar to program statements, the analysis may be modeled as a fix point computation. From this perspective, the central question can be divided into two subquestions: (1) What safety assertions should come out of the precondition of a node when its postcondition needs to discharge a given safety assertion which comes from the successor nodes of the current node. The answer can be none, which means that the postcondition of the node can discharge the given safety assertion. (2) What transformations should be applied to a node, if

the postcondition of the node cannot discharge the assertion. The first subquestion facilitates describing the analysis in the framework of abstract interpretation: the analysis reaches its fix point when there are no new safety assertions coming out of the preconditions of nodes. The second subquestion makes the analysis differ from a traditional abstract interpretation: the latter does not change a node—a program statement, while this analysis not only keeps the original node which is a judgment, but also provides enough information about how to derive new judgments for different call paths. In the framework of abstract interpretation, the computation for answering both questions can be described in the transfer functions of the analysis.

6.4.2 Abstract Domain

The domain of the analysis is the power set of all concrete safety assertions of code block judgments. The join operation is the set union, the meet operation is the set intersect, and the weaker-than relation is the set subset. A merge operation means the join operation on more than two sets.

6.4.3 Transfer Functions

There are two transfer functions in this abstract interpretation: one for code blocks, and the other for function calls. They compute an outgoing abstract state based on an incoming abstract state. An abstract state is also called a configuration, and the outgoing and incoming configurations are referred to as out-configuration and in-configuration, respectively. Each safety assertion in a configuration has some attributes, one of which is called `previousLabels`; it includes the label of the code block from which the assertion directly comes from, namely, the address of one of the successor code blocks that passes the assertion to the current code block. In the join operation, if two identical assertions come from two different successors, the new configuration only contains a single assertion, but its `previousLabels` attribute is updated to include the labels of both successors.

The two transfer functions work differently. This subsection describes the transfer function for code blocks, and the next describes the other transfer function.

6.4.3.1 Transfer Function for Code Block Nodes

The code block transfer function works as follows. For each safety assertion in the in-configuration, it tries to discharge the assertion from the label predicate entries that are in the postcondition of the node and that share some common label included in the `previousLabels` attribute of the assertion. If the discharging attempt is successful, it records the theorem that discharges the assertion in the list of discharged assertions for the node. Otherwise, it computes what transformations should be applied to the current node and what the new judgments are. Figure 6.3 gives its pseudo-code, where bnode is the current code block node, and Σ_{in} and Σ_{out} are the in- and out-configurations over all nodes inside a function, respectively.

6.4.3.1.1 Discharging methods. One of the central operations of the transfer function is to discharge a safety assertion. ARMor attempts to complete it by two methods. The first one applies the machine state at a postcondition entry to derive the safety assertion. Its implementation involves two major steps. The first step is to instantiate the free variables of the assertion with corresponding values of machine resources at the postcondition entry. The second step is trying to simplify the instantiated assertion to `true` by using simplification tactics, rules, and simpset of the metalogic. An example of using this method is the discharging of the control flow integrity assertion $\langle rf\ R14 \in \texttt{succ}(0x2C)\rangle$ of blk3 by the postcondition of blk1 discussed above.

```
transfer_block (bnode, Σ_in, Σ_out):
   foreach (assert, previousLabels) in Σ_in
      foreach (l', p) in postcondition(bnode)
         if (not (l' in previousLabels)) continue;
         if ((l', p) discharges assert) then
            store discharging theorems and the current call path;
         else
            previousLabels = {L(bnode)};
            Σ_out(bnode) = Σ_out(bnode) ∪ {(assert, previousLabels)};
            store transformation information and the current call path;
   return Σ_out;
```

Figure 6.3: Code block transfer function

The second method is used if the first one fails. It frames a branch condition of a judgment onto the judgment itself to strengthen the postcondition of the judgment. For example, the Hoare judgment of the jump branch of blk3 is

$$
\begin{array}{l}
\texttt{SPEC SAFE\_INS}\ \{(0x14, \texttt{R}\ 8\ k\ *\texttt{REG}\ rf\ * \\
\qquad \langle rf\ R14 \in \texttt{succ}(0x1C)\rangle * \langle rf\ R2 \neq 0x40000000\rangle\ * \\
\qquad \texttt{MEMORY dm}\ df * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf\ * \\
\qquad \texttt{S sC}\ c * \texttt{S sN}\ n * \texttt{S sV}\ v * \texttt{S sZ}\ z)\} \\
\quad \text{blk3} \\
\{(rf\ R14, \texttt{R}\ 8\ k\ *\texttt{REG}\ ((R1 \mapsto (rf\ R1 - 1))\ rf)\ * \\
\qquad \texttt{MEMORY dm}\ df * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf\ * \\
\qquad \texttt{S sC}\ c' * \texttt{S sN}\ n' * \texttt{S sV}\ v' * \texttt{S sZ}\ ((rf\ R2) = 0x40000000))\}.
\end{array}
\tag{6.6}
$$

When it is framed with its branch condition $\langle rf\ R2 \neq 0x40000000\rangle$, we get the following judgment with a postcondition strengthened with the branch condition:

$$
\begin{array}{l}
\texttt{SPEC SAFE\_INS}\ \{(0x14, \texttt{R}\ 8\ k\ *\texttt{REG}\ rf\ * \\
\qquad \langle rf\ R14 \in \texttt{succ}(0x1C)\rangle * \langle rf\ R2 \neq 0x40000000\rangle\ * \\
\qquad \texttt{MEMORY dm}\ df * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf\ * \\
\qquad \texttt{S sC}\ c * \texttt{S sN}\ n * \texttt{S sV}\ v * \texttt{S sZ}\ z)\} \\
\quad \text{blk3} \\
\{(rf\ R14, \texttt{R}\ 8\ k\ *\texttt{REG}\ ((R1 \mapsto (rf\ R1 - 1))\ rf)\ * \\
\qquad \langle rf\ R2 \neq 0x40000000\rangle\ * \\
\qquad \texttt{MEMORY dm}\ df * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf\ * \\
\qquad \texttt{S sC}\ c' * \texttt{S sN}\ n' * \texttt{S sV}\ v' * \texttt{S sZ}\ ((rf\ R2) = 0x40000000))\}.
\end{array}
\tag{6.7}
$$

The precondition does not change, because the branch condition term is pure, and for a pure assertion $\langle c\rangle$, we have $\langle c\rangle * \langle c\rangle = \langle c\rangle$.

Similarly, we can strengthen the postcondition of the judgment of the fall through branch of blk3, and the original and the resultant judgments are:

$$\begin{aligned}
&\texttt{SPEC SAFE_INS}\ \{(0x14, \texttt{R}\ 8\ k\ *\texttt{REG}\ rf * \langle rf\ \ R2 = 0x40000000\rangle * \\
&\qquad\qquad \texttt{MEMORY dm}\ df * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf\ * \\
&\qquad\qquad \texttt{S sC}\ c * \texttt{S sN}\ n * \texttt{S sV}\ v * \texttt{S sZ}\ z)\} \\
&\qquad\qquad \text{blk3} \qquad\qquad (6.8)\\
&\qquad \{(0x20, \texttt{R}\ 8\ k\ *\texttt{REG}\ ((R1 \mapsto (rf\ \ R1 - 1))\ rf) * \\
&\qquad\qquad \texttt{MEMORY dm}\ df * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf\ * \\
&\qquad\qquad \texttt{S sC}\ c' * \texttt{S sN}\ n' * \texttt{S sV}\ v' * \texttt{S sZ}\ ((rf\ \ R2) = 0x40000000))\}
\end{aligned}$$

and

$$\begin{aligned}
&\texttt{SPEC SAFE_INS}\ \{(0x14, \texttt{R}\ 8\ k\ *\texttt{REG}\ rf * \langle rf\ \ R2 = 0x40000000\rangle * \\
&\qquad\qquad \texttt{MEMORY dm}\ df * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf\ * \\
&\qquad\qquad \texttt{S sC}\ c * \texttt{S sN}\ n * \texttt{S sV}\ v * \texttt{S sZ}\ z)\} \\
&\qquad\qquad \text{blk3} \qquad\qquad (6.9)\\
&\qquad \{(0x20, \texttt{R}\ 8\ k\ *\texttt{REG}\ ((R1 \mapsto (rf\ \ R1 - 1))\ rf) * \\
&\qquad\qquad \langle rf\ \ R2 = 0x40000000\rangle * \\
&\qquad\qquad \texttt{MEMORY dm}\ df * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf\ * \\
&\qquad\qquad \texttt{S sC}\ c' * \texttt{S sN}\ n' * \texttt{S sV}\ v' * \texttt{S sZ}\ ((rf\ \ R2) = 0x40000000))\}.
\end{aligned}$$

With the strengthening by the branch condition, the postcondition of Judgment 6.9 is sufficient to discharge the memory safety assertion of blk4 $\langle\{rf\ \ R2\} \subseteq \texttt{dm}\rangle$, after dm is instantiated by its real value given in the safety policy of Figure 1.3(b). In this simplified example, $\texttt{dm} = \texttt{mem}$.

The second method is heuristic in nature, hoping that the enhanced postcondition might be useful. It is desirable to conduct this heuristic enhancement even if the condition is not used, because there is a Weaken rule for Hoare triples, as shown in Figure 4.3, which can always weaken the enhanced postcondition to its original form.

When either of the two methods succeeds, ARMor records the simplification theorems for the current call path. Otherwise, ARMor concludes that the safety assertion is unable to be discharged by the current node and begins a transformation process.

6.4.3.1.2 Judgment transformation. The motivation for the judgment transformation of a node is that when given a safety assertion which cannot be discharged by the postcondition of the node, some changes must be made on the node such that the postcondition can discharge the safety assertion. In the context of theorem proving, these changes must be proven correct as theorems. The basic method that ARMor uses for transformation is to use the Frame rule to add the safety assertion into the judgment of the node while maintaining the consistency of derived judgments.

Denote a postcondition entry of a node as (l, p), and let t be a safety assertion term whose `previousLabels` attribute contains l. The algorithm for judgment transformation works as follows:

1. Instantiate t with the state of (l, p) to obtain a new term t'.
2. Apply the Frame rule to add t' into the current node judgment.
3. Simplify t' in the precondition of the current node, and denote the result as t'_{pre}, while keeping t' as it is in the postcondition; this simplification process proves an equality $t' = t'_{pre}$.
4. t'_{pre} is the outgoing term for the safety assertion t, whose `previousLabels` attribute is updated to include the label of the current node with label l removed.
5. Record term t and the simplification theorem $t' = t'_{pre}$ for the current path.

There are several important implementation choices related to this algorithm, and I discuss some of the important ones.

6.4.3.1.3 Delayed transformation. The actual transformation of a node by applying the Frame rule in the 2nd step is not carried out in this stage of computation, i.e., in the abstract interpretation described here. It is postponed to the last stage

outlined in Section 6.1 and to be described in Section 6.5, for two reasons. First, it can reduce the number of proof transformations. When terms t_1 and t_2 are propagated through the current node, if the Frame rule is applied for each instantiated term t_1' and t_2', then the same proof process will be repeated twice. When this computation is delayed, only one application of the Frame rule with term $(t_1' * t_2')$ is needed.

Second, the "current node judgment" mentioned in Step 2 is the original Hoare judgment of a code block which has not gone through any transformations. This is very important to maintain the correctness of the transformation, because terms from different paths cannot be mixed together. An example illustrates this point. Suppose there is a function `add`, whose C code is as follows.

```
int add(int x, int y) {
   return x + y;
}
```

The ARM assembly code for the function is `add R0,R0,R1; mov PC,R14`, whose Hoare judgment is

$$
\begin{array}{l}
\texttt{SPEC SAFE\_INS} \\
\quad \{(p, \texttt{R}\ 8\ k * \texttt{REG}\ rf * \texttt{MEMORY dm}\ df * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf * \\
\qquad \langle rf\ \ R14 \in \texttt{succ}(p+4)\rangle)\} \\
\quad (p,\ \ \texttt{add R0,R0,R1}) \\
\quad (p+4,\ \ \texttt{mov PC,R14}) \\
\quad \{(rf\ \ R14, \texttt{R}\ 8\ k * \texttt{REG}\ ((rf\ \ R0 \mapsto (rf\ \ R0 + rf\ \ R1))\ rf) * \\
\qquad \texttt{MEMORY dm}\ df * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf)\}.
\end{array}
\tag{6.10}
$$

Assume that the function is called on two different paths. On one path, a safety assertion $\langle rf\ \ R2 > 10\rangle$ is to be discharged, and on the other path, an incompatible safety assertion $\langle rf\ \ R2 = 0\rangle$ needs to be discharged. If a transformation was done for the two assertion terms together, then the outgoing term from this judgment would have $\langle rf\ \ R2 > 10\rangle * \langle rf\ \ R2 = 0\rangle$, which is `false`. This results in a trivially true Hoare judgment, and the proof goes the wrong way.

For a correct proof solution, each path requires a unique judgment: the first one only needs $\langle rf\ \ R2 > 10\rangle$ to be framed on Judgment 6.10, and the second one only requires $\langle rf\ \ R2 = 0\rangle$ to be framed on Judgment 6.10, bringing a different judgment to each path.

The delayed transformation records all safety assertions along different paths until reaching a fix point, and those assertions belonging to the same path are framed to the original judgment of a code block to form a new judgment along the path.

6.4.3.1.4 Assertion cache. ARMor uses a cache to record what terms have been seen at a code block and what paths the term belongs to. On the second visit of the same term along a different path, ARMor only updates the paths of the term in the cache without recomputing the discharging process. When this code block level cache is combined with the function level cache, which is to be discussed later in Section 6.4.3.2.2, ARMor only processes a unique safety assertion term once.

6.4.3.2 Transfer Function for Call Nodes

As in an interprocedural abstract interpretation, a function call is modeled by a call node. What is different from a normal abstract interpretation used in a compiler or analyzer is that all reasoning processes must be proven as theorems in the HOL proof assistant. The definition of function judgment presented in Section 4.6 is based on the relationship among judgments of code blocks inside the function, and it is the only method available in $\mathcal{L}_{fn}$ to reason about a function. Therefore, the transfer function for a call must follow the logic in the definition.

A call node is an abstraction of a callee, and a discharging process must be carried out by the underlying function. The transfer function on the call node works as follows, when the node receives a safety assertion term t to discharge.

1. Set up a new fix point computation environment for the callee, such as initializing the in- and out- configurations of all nodes of the callee properly; additional, merge t into the in-configurations of all exit nodes of the callee.
2. Conduct a fix point computation on the configurations of the callee: the transfer function for code block nodes is described in Section 6.4.3.1, and the transfer

function of a call node is described here.

3. If the computation reaches a fix point, then the out-configuration of the entry node of the callee is taken as the outgoing term t' for the call node; the `previousLabels` attribute is updated to the label of the entry node.

4. There might be no term coming out of the entry node, which means that term t is discharged by some node of the callee.

5. If the computation cannot reach a fix point, then report a proof failure.

There are some implementation choices related to this algorithm to reduce redundant computation.

6.4.3.2.1 Configuration initialization. A callee may have safety assertions on its own code blocks. These assertions are initialized in the outgoing configuration when the callee function is visited for the first time. Later invocations skip the same computation on those assertions, and only the incoming safety assertions are propagated.

6.4.3.2.2 Function level assertion cache. Similar to the assertion caches used by the transfer function of code blocks, caches are also used for call nodes. For example, before the actual fix point computation is performed for a function, its cache is queried to see if t has been computed before and what its corresponding outgoing term t' is. In addition, the cache records call paths for all incoming terms. When ARMor sees the same term later, it simply updates the paths of the term at this cache, and the information for a term transformation along a path at a code block can be obtained by querying both the code block level cache and the function level cache. Because of the two caches, a safety assertion is computed only once for its transformation at a code block.

6.4.3.2.3 Failure detection. The computation may not reach a fix point for various reasons. For example, if the rewriting process used for discharging safety assertions converts a safety assertion to a new form every time, then the computation may continue for ever. ARMor currently uses the path that a term traverse to detect

this situation: if the term passes the same code block multiple times, then ARMor reports a proof failure.

6.5 Proving Function Specifications

The last step of the framework is to construct specifications of functions and to prove function judgments. Recall that the fix point computation in the previous step has recorded two pieces of information: the instantiated safety assertion terms to be framed onto the judgment of a code block along different call paths, and the equality theorems for simplifying the terms in preconditions. This step traverses through the call graph of the program, and for each function along a call path, it performs the following tasks:

1. For each code block node of the function, it conducts these steps:
   (a) Frame the safety assertion terms that belong to the call path to the judgment of the code block.
   (b) Use the equality theorems corresponding to the framed terms to simplify them in the precondition of the judgment.
   (c) Prove the framed judgment as a well-formed judgment by the definition presented in Section 4.5.3.
   (d) Apply the Base rule defined in Figure 4.4(b) to prove the well-formed judgment as a well-formed node.
2. For each call node, it conducts these steps:
   (a) Prove the function specification of the callee corresponding to the call node along the call path.
   (b) Use the Induction rule defined in Figure 4.4(b) to develop a well-formed node for the callee and use it in reasoning about the caller function.
3. Prove the implication relation, $\stackrel{P}{\Rightarrow}$, between two adjacent well-formed nodes; this is successful now, because the safety assertions of the nodes can be discharged by postconditions of its predecessor blocks with the recorded theorems.

4. Take the precondition of the entry node of the function as the initial condition, and take the union of postconditions of all exit nodes of the function as the exit condition.

5. Prove the `FUN_SPEC SAFE_INS` judgment of the function by the definition presented in Figure 4.4(a). This process reuses the discharging theorems stored by the transfer functions described in the previous subsection.

When applied to the top-level function of a program, this algorithm computes the judgment of the top-level function. By applying the judgment definition discussed in Figure 4.4(c), the program judgment can be finally proven. A proven program judgment guarantees that the program respects the memory safety and control flow integrity specifications formalized in Section 5.3.

6.6 Proof Engineering

There are practical considerations in implementing ARMor's verification framework in the HOL system. I discuss few important ones in this subsection, because it is not possible to develop a working ARMor toolchain without addressing them properly.

6.6.1 Avoiding Term Size Explosion

The middle layer of $\mathcal{L}_{fn}$ is a Hoare logic. As discussed in Section 2.2.1, Hoare logic composes judgments of bigger pieces of code from judgments of smaller pieces of code. This causes problems in the HOL proof assistant, when the size of basic blocks goes large. For example, if a basic block has 30 instructions including several store and load instructions, then the term describing the value of memory at the end of the basic block may go over 50,000 lines, which makes inspecting a judgment impossible. During ARMor's development, it was not uncommon for the proof assistant to hang when a large term was produced; the only way to get out of this situation was killing the HOL process and starting over.

A method commonly used in the HOL community is introducing unique intermediate variables for large terms. A single variable t_v is made equal to a large term

t_{large}, i.e., $t_v = t_{large}$, and in subsequent rewriting processes, the single variable t_v is used at every place where t_{large} is needed. The equality theorem $t_v = t_{large}$ may be placed in the assumption list of a theorem as an assumption.

This method does not work for ARMor's verification. In order to prove a $\overset{P}{\Rightarrow}$ relation between a pair of corresponding postcondition and precondition, ARMor needs to hide the values of machine resources by using `SEP_HIDE` in the existing ARM semantics. Because free variables are introduced at the precondition of judgments, it is possible to derive the implication relation only after values in both the postcondition and the precondition are hidden. If the values of machine resources cannot be hidden, it is impossible to deduce the implication relation. Hiding values in the postcondition may be done through the Weaken rule, because an assertion with explicit values always implies an assertion with corresponding hidden values. However, hiding values in the precondition is more difficult, since the values must be universally quantified in a judgment. Placing the equality theorem in the assumption list prevents the values from being universally quantified, because the variables that were free now become bound.

ARMor uses decomposition to overcome this problem. ARMor splits a big basic block into several smaller basic blocks, and the terms at the postcondition of each basic block are small enough so that the HOL proof assistant can handle them with ease. Because size increases caused by different instructions vary, ARMor uses empirical knowledge to choose splitting locations. A good candidate is after a store instruction, because subsequent load instructions tend to duplicate the value set by a store. Splitting after a store instruction introduces fresh free variables for memory value in the next basic block, and duplicating free variables in the next basic block is not a problem. Specifically, ARMor splits after store instructions in a big basic block, such that a basic block contains at most two store instructions.

6.6.2 Avoiding Judgment Explosion

The conditional execution of the ARM ISA presents a potential danger in composing a basic block that contains multiple conditional execution instructions. Due

to the compositional nature of the middle layer of $\mathcal{L}_{fn}$, each condition generates two judgments, so if there are n conditional execution instructions in a block, there will be 2^n judgments, resulting in a large number of judgments for a single code block. I solved this problem by splitting such a code block into multiple code blocks, such that each block only has two or four conditional execution paths. The relationship among the split code blocks is reasoned about at the top layer of $\mathcal{L}_{fn}$, which does not have this compositional issue.

6.6.3 Making Proof Units

In proving the function judgment `FUN_SPEC`, it is possible to directly use its definition to expand the term in the goal stack of HOL for a program with several basic blocks, such as the example program shown in Figure 1.3. However, this method does not scale to functions with many nodes. A direct rewriting with the definition generates a large number of subgoals, because the definition has a universal quantifier and multiple references of the same term. For a function with tens of nodes, the number of subgoals and the sizes of subgoals are collectively huge enough to make the goal stack of the proof assistant hang. The only way to get out of the hanging situation is to kill the HOL process.

To solve this problem, I grouped a node with its predecessor nodes together to form a proof unit. For each proof unit, a theorem is proven, which states that the node is well-formed, and that the postconditions of its predecessor nodes imply the precondition of the node; namely, this theorem corresponds to the last two lines of the definition shown in Figure 4.4(a). For a function with n nodes, there are n theorems. Then the collection of theorems are used as rewriting rules in expanding the definition. This technique works well for ARMor's verification. In practice, a list of over a hundred nodes can be proven without problems.

6.7 Discussion of Related Work

Reasoning about machine-code programs automatically in a higher-order logic proof assistant is challenging. At a high level, two conceptually separated processes hinder proof automation: one is the development of specifications, and the other is

proving implementation correct in terms of the specifications. Previous work has attempted to automate or semiautomate the second process. For example, Myreen et al. developed algorithms for decompiling assembly code with certain structures into logic functions and automated their proof reuse technique [77]; Ni et al. semiautomated the verification process performed in XCAP [83]; Li managed to automatically verify the correctness of the compilation of ARM code produced by the VCL compiler [61].

However, developing specifications automatically has not been addressed adequately. Nevertheless, this is very important for reasoning about machine-code programs, because they do not have high-level language structures which may facilitate specification development. It is error-prone and inefficient to write specifications manually. My work presented here is not trying to solve the general problem of developing specifications for machine-code programs; instead, it focuses on a narrow area of verifying certain safety properties. As discussed in Chapter 1, this specific area may have impacts on designing multitasking embedded systems. This framework not only automatically verifies ARM binary programs against their specifications, but also automatically generates the specifications by leveraging abstract interpretation.

Abstract interpretation has been an intensively studied area on its own since Cousot and Cousot's pioneering paper in 1977 [22]. Utilizing it to generate safety constraints has been done in some research; for example, Xu et al. used it to generate type state constraints of binary programs and then applied the VCG method to discharge those constraints [109]. Other research used it to assist theorem proving; for example, Seo et al. utilized the result from an abstract interpretation to guide the construction of Hoare logic proofs [96,97]. Their approach was approximate in nature and generated much redundant information which needed to be removed manually. In contrast, my framework tightly integrates proof analysis and abstract interpretation together and utilizes them directly generate specifications and theorems that are needed for later proof use. There is no redundant information to be removed.

The transfer function used in transforming a code block node is similar to the computation of the weakest precondition to some extent. The weakest precondition computation dates back to Dijkstra and King [24, 53]. It answers a similar question:

what is the weakest precondition for a piece of code given a postcondition. However, the weakest precondition computation in program verification does not change a node, nor does it require reasoning processes to be proven as theorems. In contrast, the precondition in ARMor's verification must be computed by applying available reasoning rules in a higher-order logic proof assistant, and whether the precondition is the weakest is irrelevant.

CHAPTER 7

ILLUSTRATION OF ARMOR'S VERIFICATION

This chapter illustrates the abstract concepts described in previous chapters; specifically, it shows the verification process of proving Theorem (1.1), which can be conducted by the ARMor toolchain automatically. The theorem guarantees the memory safety and control flow integrity of the example program shown in Figure 1.3. This process follows the steps outlined in Section 6.1.

The first step is discussed in Section 6.2, and the result after this step is the `SAFE_INS` relation presented in Section 6.2. Note that there is an instance of the relation for each state transition, proven based on the existing semantics.

The second step is described in Section 6.3, and the result after this step is the judgments listed in Table 6.1. As discussed in that section, the instantiated judgments ensure that every state of the program has assertions of the two isolation properties.

The third step is demonstrated in Section 4.5.2, and the result after this step is Hoare judgments of code blocks. Because the Hoare judgments of blk3, blk4 and blk5 have been given in Judgments (6.6), (6.8), (6.2), (6.3) and (6.4), Judgments (7.1) and (7.2) show the judgments of blk1 and blk2, respectively.

$$\begin{array}{l} \texttt{SPEC SAFE\_INS}\ \{(0x0, \texttt{R}\ 8\ k * \texttt{REG}\ rf\ * \\ \qquad\qquad \texttt{MEMORY dm}\ df * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf)\} \\ \qquad \text{blk1} \\ \{(0x14, \texttt{R}\ 8\ k * \texttt{REG}\ rf' * \\ \qquad\qquad \texttt{MEMORY dm}\ df * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf)\} \\ rf' = ((R14 \mapsto 0x10)\ ((R0 \mapsto 0)\ ((R1 \mapsto 0)\ ((R13 \mapsto 0x41000000)\ rf)))) \end{array} \tag{7.1}$$

$$\begin{array}{c} \texttt{SPEC SAFE\_INS}\ \{(0x10, p)\} \\ \text{blk2} \\ \{(0x10, p)\} \\ p = \texttt{R}\ 8\ k * \texttt{REG}\ rf * \texttt{MEMORY dm}\ df * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf \end{array} \tag{7.2}$$

One minor derivation may be done on these Hoare judgments. It is straightforward to use the Frame rule to add missing assertions to each judgment to make it global. For example, Judgments (7.1) and (7.2) are framed with the assertions of the four status flags so that they become global. I skip this proof and will assume the global version of Hoare judgments in subsequent discussion.

The fourth step runs the abstract interpretation described in Section 6.4, and the result is information about which safety assertions may be discharged at which code blocks and what transformations should apply to different code blocks. For this example, the analysis shows that the control flow integrity assertion $\langle rf\ R14 \in \texttt{succ}(0x2C)\rangle$ of code block blk5 should be framed onto the judgments of blk4 and blk3, and that it can be discharged by blk1; the memory safety assertion $\langle\{rf\ R2\} \subseteq \texttt{dm}\rangle$ of blk4 may be discharged by blk3. The detail is given in the same subsection.

The last step is constructing and proving function specifications by utilizing the results of the abstract interpretation. For example, the two global specifications $bspec_{foo}$ and $kspec_{foo}$ of function `foo` may be constructed as follows. First, we frame the control flow integrity assertion $\langle rf\ R14 \in \texttt{succ}(0x2C)\rangle$ onto judgments of blk3 and blk4. The results are shown in Judgments (7.3), (7.4) and (7.5). Although blk3 has two judgments, we only frame the judgment with the postcondition whose label goes to blk4, because this judgment is the only predecessor of blk4, and the other judgment is not. The judgment of blk4 from label $0x20$ to $0x14$ is not a direct predecessor of blk5, but it is an indirect predecessor. As a result, it is also framed with the assertion.

Two minor deduction steps are required here. The first one is to merge the judgments of the same code block together to prove a single triple for each code

$$\texttt{SPEC SAFE\_INS}\ \{(0x14, \texttt{R}\ 8\ k\ *\texttt{REG}\ rf * \langle rf\ R2 = 0x40000000\rangle *$$
$$\texttt{MEMORY dm}\ df * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf *$$
$$\texttt{S sC}\ c * \texttt{S sN}\ n * \texttt{S sV}\ v * \texttt{S sZ}\ z\ * \langle rf\ R14 \in \texttt{succ}(0x2C)\rangle)\}$$
$$\text{blk3} \tag{7.3}$$
$$\{(0x20, \texttt{R}\ 8\ k\ *\texttt{REG}\ ((R1 \mapsto (rf\ R1 - 1))\ rf) *$$
$$\texttt{MEMORY dm}\ df * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf *$$
$$\texttt{S sC}\ c' * \texttt{S sN}\ n' * \texttt{S sV}\ v' * \texttt{S sZ}\ ((rf\ R2) = 0x40000000) *$$
$$\langle rf\ R2 = 0x40000000\rangle\ * \langle rf\ R14 \in \texttt{succ}(0x2C)\rangle)\}$$

$$\texttt{SPEC SAFE\_INS}\ \{(0x20, \texttt{R}\ 8\ k\ *\texttt{REG}\ rf * \langle \{rf\ R2\} \subseteq \texttt{dm}\rangle\ * \langle rf\ R1 = 0x0\rangle *$$
$$\texttt{MEMORY dm}\ df * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf *$$
$$\texttt{S sC}\ c * \texttt{S sN}\ n * \texttt{S sV}\ v * \texttt{S sZ}\ z\ * \langle rf\ R14 \in \texttt{succ}(0x2C)\rangle)\}$$
$$\text{blk4} \tag{7.4}$$
$$\{(0x2C, \texttt{R}\ 8\ k\ *\texttt{REG}\ rf\ *\texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf *$$
$$\texttt{MEMORY dm}\ ((rf\ R2 \mapsto \texttt{w2w}(rf\ R1))\ df) *$$
$$\texttt{S sC}\ c' * \texttt{S sN}\ n' * \texttt{S sV}\ v' * \texttt{S sZ}\ ((rf\ R1) = 0x0) *$$
$$\langle rf\ R14 \in \texttt{succ}(0x2C)\rangle)\}$$

$$\texttt{SPEC SAFE\_INS}\ \{(0x20, \texttt{R}\ 8\ k\ *\texttt{REG}\ rf * \langle \{rf\ R2\} \subseteq \texttt{dm}\rangle\ * \langle rf\ R1 \neq 0x0\rangle *$$
$$\texttt{MEMORY dm}\ df * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf *$$
$$\texttt{S sC}\ c * \texttt{S sN}\ n * \texttt{S sV}\ v * \texttt{S sZ}\ z\ * \langle rf\ R14 \in \texttt{succ}(0x2C)\rangle)\}$$
$$\text{blk4} \tag{7.5}$$
$$\{(0x14, \texttt{R}\ 8\ k\ *\texttt{REG}\ rf * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf *$$
$$\texttt{MEMORY dm}\ ((rf\ R2 \mapsto \texttt{w2w}(rf\ R1))\ df) *$$
$$\texttt{S sC}\ c' * \texttt{S sN}\ n' * \texttt{S sV}\ v' * \texttt{S sZ}\ ((rf\ R1) = 0x0) *$$
$$\langle rf\ R14 \in \texttt{succ}(0x2C)\rangle)\}$$

block. This involves two inference rules. One is the Union rule as shown in Figure 4.3, e.g., Judgments (7.4) and (7.5) may be unioned into one judgment for blk4, which has two label predicates in the precondition and in the postcondition. The other

is the LPMerge rules. The first LPMerge rule merges two label predicates in the precondition of a triple. When it is applied on the unioned judgment of blk4, the two branch conditions $\langle rf\ R1 = 0x0\rangle$ and $\langle rf\ R1 \neq 0x0\rangle$ form a tautology, and a single-entry multiple-exit triple is developed below. The second deduction is to prove that these judgments are well-formed Hoare judgments by the definition given in Section 4.5.3, which is very straightforward. I will directly use the well-formed judgment relation WF_SPEC.

$$
\begin{array}{l}
\texttt{WF\_SPEC SAFE\_INS} \\
\quad \{(0x20, \texttt{R}\ 8\ k\ * \texttt{REG}\ rf * \langle\{rf\ R2\} \subseteq \texttt{dm}\rangle * \langle rf\ R14 \in \texttt{succ}(0x2C)\rangle * \\
\qquad \texttt{MEMORY dm}\ df * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf\ * \\
\qquad \texttt{S sC}\ c * \texttt{S sN}\ n * \texttt{S sV}\ v * \texttt{S sZ}\ z)\} \\
\quad \text{blk4} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (7.6) \\
\quad \{(0x14, \texttt{R}\ 8\ k\ * \texttt{REG}\ rf * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf\ * \\
\qquad \texttt{MEMORY dm}\ ((rf\ R2 \mapsto \texttt{w2w}(rf\ R1))\ df)\ * \\
\qquad \texttt{S sC}\ c' * \texttt{S sN}\ n' * \texttt{S sV}\ v' * \texttt{S sZ}\ ((rf\ R1) = 0x0)\ * \\
\qquad \langle rf\ R14 \in \texttt{succ}(0x2C)\rangle), \\
\quad\ (0x2C, \texttt{R}\ 8\ k\ * \texttt{REG}\ rf\ * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf\ * \\
\qquad \texttt{MEMORY dm}\ ((rf\ R2 \mapsto \texttt{w2w}(rf\ R1))\ df)\ * \\
\qquad \texttt{S sC}\ c' * \texttt{S sN}\ n' * \texttt{S sV}\ v' * \texttt{S sZ}\ ((rf\ R1) = 0x0)\ * \\
\qquad \langle rf\ R14 \in \texttt{succ}(0x2C)\rangle)\}.
\end{array}
$$

Similarly, the two judgments of blk3 are merged together to form a single-entry two-exit judgment, Theorem (7.7). Notice that before merging, the judgment of the true or jump branch is framed with its own branch condition in its postcondition.

Next, we apply the Base rule of the well-formed node defined in Figure 4.4(b) to get well-formed node judgments for blk3, blk4 and blk5. Because the preconditions and postconditions of these judgments are identical to the well-formed Hoare judgments shown above, I do not repeat them here. Instead, I use some notations. Let P_3, P_4, and P_5 be the preconditions of blk3, blk4 and blk5, respectively. Let Q_3, Q_4, and Q_5 be the postconditions of blk3, blk4 and blk5, respectively. What are different in

$$
\begin{array}{l}
\texttt{WF\_SPEC SAFE\_INS} \\
\quad \{(0x14, \texttt{R}\ 8\ k\ *\texttt{REG}\ rf * \langle rf\ R14 \in \texttt{succ}(0x2C)\rangle * \\
\qquad \texttt{MEMORY dm}\ df * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf\ * \\
\qquad \texttt{S sC}\ c * \texttt{S sN}\ n * \texttt{S sV}\ v * \texttt{S sZ}\ z)\} \\
\quad \text{blk3} \\
\quad \{(rf\ R14, \texttt{R}\ 8\ k\ *\texttt{REG}\ ((R1 \mapsto (rf\ R1 - 1))\ rf) * \\
\qquad \texttt{MEMORY dm}\ df * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf\ * \\
\qquad \texttt{S sC}\ c' * \texttt{S sN}\ n' * \texttt{S sV}\ v' * \texttt{S sZ}\ ((rf\ R2) = 0x40000000) * \\
\qquad \langle rf\ R2 \neq 0x40000000\rangle), \\
\quad (0x20, \texttt{R}\ 8\ k\ *\texttt{REG}\ ((R1 \mapsto (rf\ R1 - 1))\ rf) * \\
\qquad \texttt{MEMORY dm}\ df * \texttt{MEMORY cm}\ cf * \texttt{MEMORY pm}\ pf\ * \\
\qquad \texttt{S sC}\ c' * \texttt{S sN}\ n' * \texttt{S sV}\ v' * \texttt{S sZ}\ ((rf\ R2) = 0x40000000) * \\
\qquad \langle rf\ R2 = 0x40000000\rangle\ * \langle rf\ R14 \in \texttt{succ}(0x2C)\rangle)\}
\end{array}
\tag{7.7}
$$

the well-formed node judgments are that `WF_SPEC` is replaced with `WF_NODE`, and code blocks blk3, blk4 and blk5 are substituted by abstract code (`bbl` 0x14), (`bbl` 0x20), and (`bbl` 0x2C), respectively. As a result, the well-formed node judgments are:

$$
\begin{array}{ll}
\texttt{WF\_NODE SAFE\_INS}\ \{P_3\}\ (\texttt{bbl 0x14})\ \{Q_3\} & \text{// for blk3} \\
\texttt{WF\_NODE SAFE\_INS}\ \{P_4\}\ (\texttt{bbl 0x20})\ \{Q_4\} & \text{// for blk4} \\
\texttt{WF\_NODE SAFE\_INS}\ \{P_5\}\ (\texttt{bbl 0x2C})\ \{Q_5\} & \text{// for blk5.}
\end{array}
$$

From the judgments of well-formed nodes, the parameter terms of function judgment of `foo` can be constructed, and they are shown in Figure 7.1.

It is straightforward to plug these terms into the definition of function judgment and prove the judgment, because the simplification and derivation theorems needed to prove the $\stackrel{P}{\Rightarrow}$ relation between two adjacent nodes are stored at code blocks during the execution of the proof analysis in the fourth step. The proven theorem is

$$
\begin{array}{c}
\texttt{FUN\_SPEC WF\_NODE SAFE\_INS}\ prog_{foo}\ entry_{foo}\ init_{foo} \\
exits_{foo}\ predecessor_{foo}\ bspec_{foo}\ kspec_{foo}
\end{array}
$$

$$
\begin{aligned}
prog_{foo} &= \{(\texttt{bbl 0x14}), (\texttt{bbl 0x20}), (\texttt{bbl 0x2C})\} \\
entry_{foo} &= \texttt{fst } P_3 \\
init_{foo} &= \texttt{snd } P_3 \\
exits_{foo} &= \{(\texttt{bbl 0x14},\ Q_3),\ (\texttt{bbl 0x2C},\ Q_5)\} \\
predecessor_{foo} &= [(\texttt{bbl 0x14}) \mapsto \{(\texttt{bbl 0x20})\}, \\
&\qquad (\texttt{bbl 0x20}) \mapsto \{(\texttt{bbl 0x14})\}, \\
&\qquad (\texttt{bbl 0x2C}) \mapsto \{(\texttt{bbl 0x20})\}] \\
bspec_{foo} &= [(\texttt{bbl 0x14}) \mapsto P_3,\ (\texttt{bbl 0x20}) \mapsto P_4,\ (\texttt{bbl 0x2C}) \mapsto P_5] \\
kspec_{foo} &= [(\texttt{bbl 0x14}) \mapsto Q_3,\ (\texttt{bbl 0x20}) \mapsto Q_4,\ (\texttt{bbl 0x2C}) \mapsto Q_5],
\end{aligned}
$$

where `fst` and `snd` returns the first and second element of a tuple, respectively.

Figure 7.1: Function specifications of `foo`

where `WF_NODE` marks that every node in the function is well-formed, and `SAFE_INS` indicates that every state of the program has assertions of memory safety and control flow integrity.

By applying the Induction rule defined in Figure 4.4(b) on the above theorem, a well-formed node for function `foo` is developed:

$$\texttt{WF\_NODE SAFE\_INS } \{P_3\}\ (\texttt{fun 0x14})\ \{Q_3 \cup Q_5\}. \tag{7.8}$$

This judgment looks just like a normal Hoare triple in syntax; interpreted by its semantics defined in Figure 4.4 and the semantics of `SAFE_INS`, it states that every node in the function stored at address `0x14` is well-formed, that the precondition of a node is implied by the postconditions of its predecessor nodes, that if the precondition of P_3 is satisfied, then postcondition of $(Q_3 \cup Q_5)$ is also satisfied, and that under the given precondition, the judgment guarantees the memory safety and the control flow integrity at every state of the function.

The precondition P_3 has an undischarged control flow integrity assertion $\langle rf\ R14 \in \texttt{succ}(0x2C)\rangle$ as shown in Judgment (7.7). It should be discharged in the caller of the function.

After developing Judgment (7.8), there are three well-formed nodes to reason about in the caller `entryFun` as shown in Figure 1.4(b). The local version of Hoare

judgments of blk1 and blk2 is shown as Judgments (7.1) and (7.2). After they are framed with missing assertions to become global, we repeat the fourth and the fifth step of the framework for function `entryFun`, as we just did for function `foo`.

The abstract interpretation finds out that the postcondition of blk1, where R14 is 0x10, discharges the assertion $\langle rf\ \ R14 \in \texttt{succ}(0x2C)\rangle$, because the given safety policy in Figure 1.3(b) has $\texttt{succ}(0x2C) = \{0x10\}$. The derivation theorems are saved at blk1 during the analysis.

After the analysis, the needed deductions discussed above are performed, such as proving well-formed Hoare judgments of blk1 and blk2 and converting them to well-formed nodes. For example, let P_1 and P_2 be the preconditions of the well-formed nodes for blk1 and blk2, respectively; let Q_1 and Q_2 be the postconditions of the well-formed nodes for blk1 and blk2, respectively. We can write $P_1 = \{(0x0, (\lambda s.\ \texttt{T}))\}$, if we focus on the predicate of state contents, not on the contents themselves. Then the well-formed node judgments for blk1 and blk2 are

$$\texttt{WF\_NODE SAFE\_INS}\ \{P_1\}\ (\texttt{bbl 0x0})\ \{Q_1\} \qquad \text{// for blk1}$$
$$\texttt{WF\_NODE SAFE\_INS}\ \{P_2\}\ (\texttt{bbl 0x10})\ \{Q_2\} \qquad \text{// for blk2.}$$

Together with Judgment (7.8), they comprise the well-formed nodes of `entryFun`. Let $P_{foo} = P3$, and $Q_{foo} = \{Q3 \cup Q5\}$. The function specifications of `entryFun` can be constructed and given in Figure 7.2.

Similarly, it is straightforward to prove the function judgment of `entryFun` as the following theorem based on the result of the proof analysis.

$$\texttt{FUN\_SPEC WF\_NODE SAFE\_INS}\ prog_{entryFun}\ entry_{entryFun}\ init_{entryFun}$$
$$exits_{entryFun}\ predecessor_{entryFun}\ bspec_{entryFun}\ kspec_{entryFun}$$

It says that every node in the function stored at address `0x0` is well-formed, that the precondition of a node is implied by the postconditions of its predecessor nodes, and that the judgment guarantees the memory safety and the control flow integrity at every state of the function.

After proving the judgment for the top-level function, the last minor step to reach Theorem (1.1) is to existentially quantify the exit condition $exits_{entryFun}$ and the

$$
\begin{aligned}
prog_{entryFun} &= \{(\texttt{bbl 0x0}), (\texttt{fun 0x14}), (\texttt{bbl 0x10})\} \\
entry_{entryFun} &= \texttt{0x0} \\
init_{entryFun} &= (\lambda s.\ \texttt{T}) \\
exits_{entryFun} &= \{(\texttt{bbl 0x10},\ Q_2)\} \\
predecessor_{entryFun} &= [(\texttt{bbl 0x0}) \mapsto \{\}, \\
&\qquad (\texttt{fun 0x14}) \mapsto \{(\texttt{bbl 0x0})\}, \\
&\qquad (\texttt{bbl 0x10}) \mapsto \{(\texttt{fun 0x14}),\ (\texttt{bbl 0x10})\}] \\
bspec_{entryFun} &= [(\texttt{bbl 0x0}) \mapsto P_1,\ (\texttt{fun 0x14}) \mapsto P_{foo},\ (\texttt{bbl 0x10}) \mapsto P_2] \\
kspec_{entryFun} &= [(\texttt{bbl 0x0}) \mapsto Q_1,\ (\texttt{bbl 0x14}) \mapsto Q_{foo},\ (\texttt{bbl 0x10}) \mapsto Q_2].
\end{aligned}
$$

Figure 7.2: Function specifications of `entryFun`

continuation specification $kspec_{entryFun}$. It is easy to prove the following program judgment by the definition given in Figure 4.4(c), and it is the final result presented in Theorem (1.1).

`PROG_SPEC SAFE_INS` $prog_{entryFun}\ entry_{entryFun}\ predecessor_{entryFun}\ bspec_{entryFun}$.

CHAPTER 8

IMPLEMENTATION AND RESULTS

I implemented the SFI mechanisms described in Chapter 3 in C using the Diablo platform; $\mathcal{L}_{fn}$ and ARMor's verification framework are implemented in the HOL system: the definitions of $\mathcal{L}_{fn}$ and the formalization of the isolation properties are defined in the metalogic, and the proof analysis and other algorithms are implemented in SML, which is the programming environment of HOL. The C code is 2,500 lines long, total HOL/SML code is 11,500 lines long. Among the HOL/SML code, only 58 lines are logic definitions and formalization of the isolation properties, about 800 lines are scripts for proving inference rules and useful theorems, and the rest implements abstract interpretation, proof of function judgments and other supporting libraries. In addition, there are 850 lines of Perl scripts that automate testing for the SFI implementation, extract the output of Diablo so that program text and data can be input into the HOL system.

Although C code and SML code are developed, neither of them is trusted. What is trusted is the formal definitions in the metalogic; ARMor's reasoning process is guaranteed by the proof assistant in the form of theorems. If there are errors in the SFI implementation or in the abstract interpretation, proving a function judgment will fail. The purpose of the SFI implementation is to provide necessary invariants that make the proof succeed; without it or with a buggy implementation, the proof will simply fail. The purpose of the abstract interpretation is to automate the discovery of the function specifications that define the function judgment; without it, depending on human efforts to find global invariants in machine code is daunting and very inefficient, if possible.

I applied ARMor to automatically prove the memory safety and control flow integrity properties of ARM executables including my test programs and MiBench

programs [44]. The proven MiBench programs are BitCount and StringSearch. BitCount has 293 machine words in its code section, and StringSearch has 1104 machine words in its code section. It took 2.5 hours to prove BitCount and 8 hours to prove StringSearch on a 2.7 GHz Core i7 machine. These programs are compiled with GCC 3.3.2 with optimization level `-Os` and run on a development board based on a Philips LPC2129 processor, which implements the ARM7TDMI architecture. To the best of my knowledge, this is the first time that realistic programs have been automatically verified in a high-order logic proof assistant, providing the highest level guarantee that can be achieved by today's computer technologies.

8.1 Trusted Computing Base

The TCB provided by ARMor includes the formalization of isolation properties, the definitions of $\mathcal{L}_{fn}$, the formal semantics of ARM ISA, the HOL proof assistant and hardware. Among them, I contributed the first two, whose definitions are 58 lines in HOL.

I compare ARMor with other work that uses sandboxing techniques to isolate untrusted binary code such as Gleipnir [1,28], PittSFIeld [66], and Native Client [95,111] in Table 8.1. Some of the projects are quite big, and I only compare the sandboxing parts in terms of the size of TCB and verification methods used.

The Gleipnir project developed CFI and XFI. For CFI, it performed theoretical analysis at an assembly language level [2], describing formal semantics for a simplified instruction set and for attack models, with final theorems establishing the correctness of its mechanisms. However, this work was checked with human endeavor by pen and paper, which means that any implementation must be trusted. XFI used a static verifier to check the presence of CFI and memory guards. The verifier is a 3000-line C++ program, which brings itself and a compiler into its TCB. PittSFIeld also used a verifier, but as an improvement in verification, it formalized semantics in ACL2 for a very small subset of instructions and for the verifier constraints; under the semantics and constraints, it proved mechanically that its mechanisms could guarantee the confinement of untrusted code [64]. NativeClient also relies on its verifier to ensure

Table 8.1: Comparison of TCBs and formal verification

	Gleipnir (CFI/XFI)	PittsFIeld	NativeClient	ARMor
TCB	3000-line commented C++ (XFI), compiler	500-line ACL2, N/A for verifier, compiler	600 C statements for x86, unknown for ARM & x86-64, compiler	58-line formal definitions of safety properties and logic, ARM semantics and HOL
Formal methods	human-checked proof at language level	machine-checked proof at language level	N/A	automatic machine-checked proof
Formalized elements	semantics of small subset of instructions and attack models	semantics of small subset of instructions and verifier constraints	N/A	safety properties and $\mathcal{L}_{fn}$, plus existing ARM semantics

safety, and many testing efforts were made to ensure the correctness of the verifier. Its verifier has 600 C statements for x86, and the sizes of verifiers for the ARM and x86-64 architectures were not reported.

None of the previous projects has verified any realistic program at the binary level. It is only ARMor that ensures formalized safety properties in binary code with an automatic machine-checked proof, and this insurance is deeply rooted in a formal realistic semantics of the ARM ISA.

8.2 Influence of Formalization

Formalizing safety properties and proving them for an executable expose a large amount of information about the executable, which reveals useful knowledge about verifying the properties and helps to correct errors in the SFI implementation.

8.2.1 Simplifying Proof

Initially, proving memory safety and control flow integrity was thought of as a tricky process, because they mutually depend on each other: the memory safety needs the control flow integrity to ensure that the store checks cannot be circumvented, while the control flow integrity depends on the memory safety to guarantee that the memory locations storing jump targets are not overwritten. In practice, introducing the control stack and giving a smaller writable data memory set `dm`, which is discussed in Section 5.2, are strong enough to prove the control flow integrity for most code cases in ARM executables. For example, targets of switch jumps are stored in the datapool, which is not included in the set of given writable addresses. As a result, after the datapool memory is formalized as an independent and constant heap assertion, the control flow integrity of switch statements can be proven. As another example, the problem of overwriting return addresses is solved by introducing the control stack. The solution of function pointers depends on how the pointers are used. If they are not meant to be changed after being placed in a table, the addresses storing them may be excluded from the given set of writable addresses, and they may be handled the same way as for switch jumps. If the function pointers are allowed to be overwritten with different values, a more complicated proof scheme is needed. So far, I have not

considered this situation in ARMor's implementation.

8.2.2 Locating Errors in SFI Implementation

ARMor is not designed to find bugs, but the failure of a proof reveals useful information about possible issues in binary code. An example is that the link register, R14, in the ARM ISA may be used as a scratch register in computation. My initial implementation of the control stack only considered loading values into PC. As a result, the control flow integrity assertion failed, when R14 was used as a scratch register and later on loaded a return address. By looking at the values used in the assertion and that of the R14, I found the reason of failure and considered instructions that load a value into R14.

8.2.3 Removing Unnecessary Checks

Dynamic checks are inserted into an ARM executable to provide necessary invariants for verification. ARMor's proof reveals that not all stores need checks. One example is storing the control stack pointer, when there are not recursive functions in a program, ARMor is able to prove the safe program judgment without adding checks for the control stack pointer. This is the current implementation in ARMor's rewriting, given that most embedded programs do not use recursion.

8.3 Overhead of Safety Checks

I measured the performance overhead of ARMor's SFI implementation for the programs proven, and it ranges from 5% to 240%. For example, BitCount has 10% slowdown, and StringSearch has 240% slowdown. The high overhead is caused by the address checking routine described in Section 3.2.1, because it is a rather lengthy function with several load and comparison instructions. This suboptimal implementation is used, because my research goal is to provide a very high-confidence argument for strict memory safety and strict control flow integrity about binary code, not to reduce overhead; this routine is the most direct way to implement a check. In addition, the implementation is not optimized. If alternative SFI implementations were used with less strict safety policies, the overhead would be reduced dramatically

as illustrated in [105] and [95], but ARMor would verify a less stringent safety requirement.

CHAPTER 9

CONCLUSION AND FUTURE WORK

This dissertation has answered this question: what is the minimal TCB for an isolation service based on SFI techniques for small multitasking embedded systems? The TCB achieved by this dissertation includes just the formal definitions of isolation properties, instruction semantics, program logic, and the proof assistant, besides hardware. It does not include a compiler, an assembler, a verifier, a rewriter, or an operating system. To the best of my knowledge, this is the smallest TCB that has ever been shown for guaranteeing nontrivial properties of realistic binary programs.

This is achieved by combining SFI techniques and high-confidence formal verification. An SFI implementation inserts dynamic checks before dangerous operations, and these checks provide necessary invariants needed by the formal verification to prove theorems about the isolation properties of ARM binary programs. The high-confidence of the formal verification is built on two facts. First, the verification is based on an existing realistic semantics of the ARM ISA that is independently developed by Cambridge researchers. Second, the verification is conducted in a higher-order mechanized proof assistant—the HOL theorem prover.

In addition, the entire verification process, including both specification generation and specification verification, is completed automatically in the proof assistant. To support proof automation, a novel program logic has been designed, and an automatic reasoning framework for verifying shallow safety properties has been developed. The program logic integrates Hoare-style reasoning and Floyd's inductive assertion reasoning together in a small set of logic definitions, which overcome shortcomings of Hoare logic and facilitate proof automation. All inference rules are proven based on the instruction semantics and the logic definitions. The reasoning framework leverages abstract interpretation to automatically find function specifications required by the

program logic. All these techniques work in concert to create the smallest TCB successfully.

9.1 Future Work

Using a high-order interactive proof assistant such as HOL has some fundamental limitations in automatic proofs. First, there lacks support of effective decision procedures. Proof automation must be done through tactics, and reasoning about program facts can only be done through SML programming. Second, all deductions are implemented by term rewriting, and rewriting in HOL is very costly in performance. For example, in deciding if a safety assertion can be derived from the postcondition of a node, it is not uncommon to take several seconds to obtain a success or failure result, and the time amounts to hours for the entire analysis of a program. To addresses these limitations, some research has been started to make proof automation easier in the HOL system. For example, Fox integrated a SAT solver into the reasoning process of word expressions [36]. Tjark and other researchers have worked on integrating the Yices and Z3 SMT solvers into the HOL proof assistant [12, 57, 107]. The proof assistant delegates constraints to an SMT solver and constructs a proof based on the answer from the solver, so that automatic reasoning about constraints can be made more effective. A future direction is to explore these features in ARMor's verification.

REFERENCES

[1] Abadi, M., Budiu, M., Erlingsson, Ú., and Ligatti, J. Control flow integrity: Principles, implementations, and applications. In *Proc. of the 12th ACM Conf. on Computer and Communications Security (CCS)* (Alexandria, VA, Nov. 2005).

[2] Abadi, M., Budiu, M., Erlingsson, Ú., and Ligatti, J. A theory of secure control flow. In *Intl. Conf. on Formal Engineering Methods* (2005), pp. 111–124.

[3] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine 7*, 49 (Nov. 1996).

[4] Ansel, J., Marchenko, P., Erlingsson, Ú., Taylor, E., Chen, B., Schuff, D., Sehr, D., Biffle, C. L., and Yee, B. S. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *Proc. of the ACM SIGPLAN 2011 Conf. on Programming Language Design and Implementation (PLDI)* (2011).

[5] Appel, A. W. Foundational proof-carrying code. In *Proc. of the 16th IEEE Symp. on Logic in Computer Science (LICS)* (Washington, DC, USA, June 2001), pp. 247–256.

[6] Appel, A. W., and Felty, A. P. A semantic model of types and machine instructions for proof-carrying code. In *Proc. of the 27th Symp. on Principles of Programming Languages (POPL)* (2000), pp. 243–253.

[7] ARM Ltd. *ARM Architecture Reference Manual*, 2005.

[8] Benton, N. A typed, compositional logic for a stack-based abstract machine. In *Programming Languages and Systems* (2005), vol. 3780 of *LNCS*, pp. 364–380.

[9] Bhatkar, E., Duvarney, D. C., and Sekar, R. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proc. of the 12th USENIX Security Symp.* (2003), pp. 105–120.

[10] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. A static analyzer for large safety-critical software. In *Proc. of the ACM SIGPLAN 2003 Conf. on Programming Language Design and Implementation (PLDI)* (San Diego, CA, June 2003).

[11] Blazy, S., Dargaye, Z., and Leroy, X. Formal verification of a C compiler front-end. In *Proc. of the Intl. Conf. on Formal Methods* (2006), pp. 460–475.

[12] Böhme, S., Fox, A., Sewell, T., and Weber, T. Reconstruction of Z3's bit-vector proofs in HOL4 and Isabelle/HOL. In *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings* (2011), J.-P. Jouannaud and Z. Shao, Eds., vol. 7086 of *Lecture Notes in Computer Science*, Springer, pp. 183–198.

[13] Boyer, R. S., and Yu, Y. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM 43* (Jan. 1996), 166–192.

[14] Buchanan, E., Roemer, R., Shacham, H., and Savage, S. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proc. of the 15th ACM Conf. on Computer and Communications Security (CCS)* (2008), pp. 27–38.

[15] Bulba, and Kil34. Bypassing stackguard and stackshield. *Phrack Magazine 10*, 56 (May 2000).

[16] Chanet, D., Sutter, B. D., Bus, B. D., Put, L. V., and Bosschere, K. D. Automated reduction of the memory footprint of the Linux kernel. *ACM Transactions on Embedded Computing Systems 6*, 4 (9 2007), 23.

[17] Cheney, C. J. A nonrecursive list compacting algorithm. *Communications of the ACM 13*, 11 (Nov. 1970), 677–678.

[18] Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic 5* (1940), 56–68.

[19] Cifuentes, C., Lewis, B., and Ung, D. Walkabout: A retargetable dynamic binary translation framework. Tech. rep., Sun Microsystems, Inc., Mountain View, CA, USA, 2002.

[20] Colby, C., Lee, P., Necula, G. C., Blau, F., Plesko, M., and Cline, K. A certifying compiler for Java. In *Proc. of the ACM SIGPLAN 2000 Conf. on Programming Language Design and Implementation (PLDI)* (Vancouver, Canada, June 2000), pp. 95–107.

[21] Cousot, P. Proving the absence of run-time errors in safety-critical avionics code. In *Proc. of the 7th Intl. Conf. on Embedded Software (EMSOFT)* (Oct. 2007).

[22] Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the 4th Symp. on Principles of Programming Languages (POPL)* (Los Angeles, CA, Jan. 1977), pp. 238–252.

[23] Department of Defense. Department of defense trusted computer system evaluation criteria, 1985. DoD 5200.28-STD.

[24] Dijkstra, E. W. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM 18*, 8 (Aug. 1975), 453–457.

[25] Dijkstra, E. W. *A Discipline of Programming*. Prentice-Hall, 1976.

[26] Emami, M., Ghiya, R., and Hendren, L. J. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proc. of the ACM SIGPLAN 1994 Conf. on Programming Language Design and Implementation (PLDI)* (1994), pp. 242–256.

[27] Emden, M. H. V. Programming with verification conditions. *IEEE Transactions on Software Engineering 5* (1979), 148–159.

[28] Erlingsson, Ú., Abadi, M., Vrable, M., Budiu, M., and Necula, G. C. XFI: Software guards for system address spaces. In *Symp. on Operating Systems Design and Implementation (OSDI)* (2006).

[29] Erlingsson, Ú., and Schneider, F. B. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop* (1999), pp. 87–95.

[30] Feng, X., Shao, Z., Dong, Y., and Guo, Y. Certifying low-level programs with hardware interrupts and preemptive threads. In *Proc. of the ACM SIGPLAN 2008 Conf. on Programming Language Design and Implementation (PLDI)* (Tucson, AZ, June 2008).

[31] Feng, X., Shao, Z., Vaynberg, A., Xiang, S., and Ni, Z. Modular verification of assembly code with stack-based control abstractions. In *Proc. of the ACM SIGPLAN 2006 Conf. on Programming Language Design and Implementation (PLDI)* (June 2006), pp. 401–414.

[32] Flanagan, C., and Saxe, J. B. Avoiding exponential explosion: Generating compact verification conditions. In *Proc. of the 28th Symp. on Principles of Programming Languages (POPL)* (2001), pp. 193–205.

[33] Floyd, R. W. Assigning meaning to programs. In *Mathematical Aspects of Computer Science* (1967), vol. 19, pp. 19–32.

[34] Ford, B., and Cox, R. Vx32: Lightweight user-level sandboxing on the x86. In *Proc. of the 2008 USENIX Annual Technical Conf.* (Boston, Massachusetts, USA, June 2008), pp. 293–306.

[35] Fox, A. Formal specification and verification of ARM6. In *Proc. of the 16th Intl. Conf. on Theorem Proving in Higher Order Logics (TPHOLs)* (Rome, Italy, Sept. 2003), pp. 25–40.

[36] Fox, A. LCF-style bit-blasting in HOL4. In *Proc. of the Intl. Conf. on Interactive Theorem Proving (ITP)* (2011).

[37] Fox, A., and Myreen, M. O. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *Proc. of the Intl. Conf. on Interactive Theorem Proving (ITP)* (Edinburgh, UK, July 2010).

[38] Goldberg, I., Wagner, D., Thomas, R., and Brewer, E. A. A secure environment for untrusted helper applications—confining the wily hacker. In *Proc. of the 6th USENIX Security Symp.* (1996).

[39] Gordon, M. *From LCF to HOL: A Short History.* MIT Press, Cambridge, MA, USA, 2000, pp. 169–185.

[40] Gordon, M. J. C. Mechanizing programming logics in higher order logic. In *Current Trends in Hardware Verification and Automated Theorem Proving*, G. Birtwistle and P. A. Subrahmanyam, Eds. Springer-Verlag, 1989, pp. 387–439.

[41] Gordon, M. J. C. *A Mechanized Hoare Logic of State Transitions.* Prentice Hall International (UK) Ltd., 1994, pp. 143–159.

[42] Gordon, M. J. C., and Melham, T. F., Eds. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[43] Greve, D., Richards, R., and Wilding, M. A summary of intrinsic partitioning verification. In *Proc. of the Fifth Intl. Workshop on the ACL2 Theorem Prover and Its Applications* (2004).

[44] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. of the of Workshop on Workload Characterization* (Austin, TX, Dec. 2001), pp. 3–14. `http://www.eecs.umich.edu/mibench`.

[45] Hardin, D. S. A robust machine code proof framework for highly secure applications. In *Proc. of the 2006 ACL2 Workshop* (2006).

[46] Hoare, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM 12*, 10 (Oct. 1969), 576–580.

[47] HOL Documentation. The HOL system logic. `http://hol.sourceforge.net/documentation.html`.

[48] Homeier, P. V., and Martin, D. F. A mechanically verified verification condition generator, July 1995.

[49] Huneycutt, C., and Mackenzie, K. Software caching using dynamic binary rewriting for embedded devices. In *Proc. of the Intl. Conf. on Parallel Processing* (2001), pp. 621–630.

[50] Jager, I., Avgerinos, T., Schwartz, E., and Brumley, D. Bap: Binary analysis platform. In *Computer Aided Verification (CAV)* (2011).

[51] Jhala, R., and Majumdar, R. Interprocedural analysis of asynchronous programs. In *Proc. of the 34th Symp. on Principles of Programming Languages (POPL)* (Nice, France, Jan. 2007), pp. 339–350.

[52] Kaufmann, M., Boyer, R. S., and Moore, J. S. The Boyer-Moore theorem prover and its interactive enhancement. *Computers and Mathematics with Applications 29*, 2 (1995), 27–62.

[53] King, J. C. *A Program Verifier*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1970.

[54] King, S., Hammond, J., Chapman, R., Pryor, A., King, S., Hammond, J., and Chapman, R. Is proof more cost effective than testing? *IEEE Transactions on Software Engineering 26* (2000), 675–686.

[55] Klein, G. Operating system verification—an overview, June 2008.

[56] Kohli, P., and Bruhadeshwar, B. FormatShield: A binary rewriting defense against format string attacks. In *Information Security and Privacy* (2008), vol. 5107 of *LNCS*, pp. 376–390.

[57] Kumar, R., and Weber, T. Validating QBF validity in HOL4. In *Proc. of the Intl. Conf. on Interactive Theorem Proving (ITP)* (Aug. 2011), M. C. J. D. van Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, Eds., vol. 6898 of *Lecture Notes in Computer Science*, Springer, pp. 168–183.

[58] Larus, J. R., and Ball, T. Rewriting executable files to measure program behavior. *Software Practice and Experience 24* (1994), 197–218.

[59] Leino, K. R. M. Efficient weakest preconditions. *Information Processing Letters 93* (Mar. 2005).

[60] Leroy, X. Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant. In *Proc. of the 33rd Symp. on Principles of Programming Languages (POPL)* (2006).

[61] Li, G. Validated compilation through logic. In *Proc. of the 17th Intl. Conf. on Formal Methods* (2011), pp. 169–183.

[62] Linn, C., and Debray, S. Obfuscation of executable code to improve resistance to static disassembly. In *ACM Conf. on Computer and Communications Security (CCS)* (2003), pp. 290–299.

[63] Matthews, J., Moore, J. S., Ray, S., and Vroon, D. Verification condition generation via theorem proving. In *Proc. of the 13th Intl. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)* (Phnom Penh, Cambodia, Nov. 2006), pp. 362–376.

[64] McCamant, S. A machine-checked safety proof for a CISC-compatible SFI technique. In *MIT CSAIL Technical Report* (May 2006).

[65] McCamant, S., and Morrisett, G. Efficient, verifiable binary sandboxing for a CISC architecture. In *MIT CSAIL Technical Report* (May 2005).

[66] McCamant, S., and Morrisett, G. Evaluating SFI for a CISC architecture. In *Proc. of the 15th USENIX Security Symp.* (Aug. 2006).

[67] McCarthy, J. Towards a mathematical science of computation. In *Intl. Federation for Information Processing Congress* (1962).

[68] Melham, T. F. A package for inductive relation definitions in HOL. In *Proc. of the 1991 Intl. Workshop on the HOL Theorem Proving System and its Applications* (1992), pp. 350–357.

[69] Milner, R. Logic for computable functions: Description of a machine implementation. Tech. rep., Stanford University, Stanford, CA, USA, 1972.

[70] Morrisett, G., Tan, G., Tassarotti, J., Tristan, J.-B., and Gan, E. RockSalt: Better, faster, stronger SFI for the x86. In *Proc. of the ACM SIGPLAN 2012 Conf. on Programming Language Design and Implementation (PLDI)* (June 2012).

[71] Morrisett, G., Walker, D., Crary, K., and Glew, N. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems 21*, 3 (May 1999), 527–568.

[72] Muth, R., Debray, S., Watterson, S., Bosschere, K. D., and Informatiesystemen, V. E. E. Alto: A link-time optimizer for the Compaq Alpha. *Software - Practice and Experience 31* (1999), 67–101.

[73] Myreen, M. O. *Formal Verification of Machine-Code Programs*. PhD thesis, University of Cambridge, Dec. 2008.

[74] Myreen, M. O. Verified just-in-time compiler on x86. In *Proc. of the 37th Symp. on Principles of Programming Languages (POPL)* (Jan. 2010), pp. 107–118.

[75] Myreen, M. O., Fox, A. C. J., and Gordon, M. J. C. A Hoare logic for ARM machine code. In *Proc. of the IPM Intl. Symp. on Fundamentals of Software Engineering (FSEN)* (2007).

[76] Myreen, M. O., and Gordon, M. J. C. A Hoare logic for realistically modeled machine code. In *Proc. of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2007), pp. 568–582.

[77] Myreen, M. O., Slind, K., and Gordon, M. J. C. Machine-code verification for multiple architectures—An application of decompilation into logic. In *Proc. of the Intl. Conf. on Formal Methods in Computer-Aided Design* (2008).

[78] Native Client. http://code.google.com/p/nativeclient/issues/detail?id=245.

[79] Necula, G. C. Proof-carrying code. In *Proc. of the 24th Symp. on Principles of Programming Languages (POPL)* (Paris, France, Jan. 1997), pp. 106–119.

[80] Necula, G. C., and Lee, P. Safe kernel extensions without run-time checking. In *Proc. of the 2nd Symp. on Operating Systems Design and Implementation (OSDI)* (1996), pp. 229–243.

[81] Nethercote, N., and Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proc. of the ACM SIGPLAN 2007 Conf. on Programming Language Design and Implementation (PLDI)* (June 2007).

[82] Ni, Z. *Modular Machine Code Verification*. PhD thesis, Yale University, May 2007.

[83] Ni, Z., and Shao, Z. Certified assembly programming with embedded code pointers. In *Proc. of the 33rd Symp. on Principles of Programming Languages (POPL)* (Charleston, SC, USA, Jan. 2006), pp. 320–333.

[84] Olszewski, M., Cutler, J., and Steffan, J. G. Judostm: A dynamic binary-rewriting approach to software transactional memory. In *Proc. of the 16th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)* (Brasov, Romania, Sept. 2007).

[85] Owens, S., Sarkar, S., and Sewell, P. A better x86 memory model: x86-TSO. In *Proc. of the 22nd Intl. Conf. on Theorem Proving in Higher Order Logics (TPHOLs)* (2009), pp. 391–407.

[86] Pierro, A. D., and Wiklicky, H. Measuring the precision of abstract interpretations. In *Proc. of the Intl. Workshop on Logic Based Program Synthesis and Transformation (LOPSTR)* (London, UK, July 2001), Springer-Verlag, pp. 147–164.

[87] Prasad, M., and cker Chiueh, T. A binary rewriting defense against stack based overflow attacks. In *Proc. of the USENIX Annual Technical Conf.* (2003), pp. 211–224.

[88] Put, L. V., Chanet, D., and Bosschere, K. D. Whole-program linear-constant analysis with applications to link-time optimization. In *Proc. of the 10th Intl. Workshop on Software and Compilers for Embedded Systems* (Nice, France, Apr. 2007), pp. 61–70.

[89] Ray, S., Hao, K., Chen, Y., Xie, F., and Yang, J. Formal verification for high-assurance behavioral synthesis. In *Proc. of the 7th Intl. Symp. on Automated Technology for Verification and Analysis (ATVA)* (Macao, China, Oct. 2009), pp. 337–351.

[90] Reynolds, J. C. Separation logic: A logic for shared mutable data structures. In *Proc. of the 17th IEEE Symp. on Logic in Computer Science (LICS)* (2002), pp. 55–74.

[91] Saabas, A., and Uustalu, T. A compositional natural semantics and Hoare logic for low-level languages. *Theoretical Computer Science 373*, 3 (Mar. 2007), 273–302.

[92] Saltzer, J. H., and Schroeder, M. D. The protection of information in computer systems. *Proceedings of the IEEE 9*, 63 (Sept. 1975).

[93] Schirmer, N. A verification environment for sequential imperative programs in Isabelle/HOL. In *Logic for Programming, AI, and Reasoning* (2005), vol. 3452, Springer, pp. 398–414.

[94] Schneider, F. B., Morrisett, J. G., and Harper, R. A language-based approach to security. In *Informatics—10 Years Back, 10 Years Ahead* (2001), pp. 86–101.

[95] Sehr, D., Muth, R., Biffle, C., Khimenko, V., Pasko, E., Schimpf, K., Yee, B., and Chen, B. Adapting software fault isolation to contemporary CPU architectures. In *Proc. of the 19th USENIX Security Symp.* (Aug. 2010).

[96] Seo, S., Yang, H., and Yi, K. Automatic construction of Hoare proofs from abstract interpretation results. In *Proc. of the 1st Asian Symp. on Programming Languages and Systems* (2003), vol. 2895 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 230–245.

[97] Seo, S., Yang, H., Yi, K., and Han, T. Goal-directed weakening of abstract interpretation results. *ACM Transactions on Programming Languages and Systems 29*, 6 (Oct. 2007).

[98] Shacham, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proc. of the 14th ACM Conf. on Computer and Communications Security (CCS)* (2007).

[99] Sites, R. L., Chernoff, A., Kirk, M. B., Marks, M. P., and Robinson, S. G. Binary translation. *Communications of the ACM 36* (Feb. 1993).

[100] Small, C. A tool for constructing safe extensible C++ systems. In *Proc. of the 3rd USENIX Conf. on Object-Oriented Technologies* (1997).

[101] Sutter, B. D., Put, L. V., Chanet, D., Bus, B. D., and Bosschere, K. D. Link-time compaction and optimization of ARM executables. *ACM Transactions on Embedded Computing Systems (TECS) 6* (Feb. 2007).

[102] Tan, G. *A Compositional Logic for Control Flow and its Application in Foundational Proof-Carrying Code*. PhD thesis, Princeton University, Sept. 2006.

[103] Tan, G., and Appel, A. W. A compositional logic for control flow. In *Proc. of the 7th Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI)* (2006), pp. 80–94.

[104] Van Put, L., Chanet, D., De Bus, B., De Sutter, B., and De Bosschere, K. DIABLO: A reliable, retargetable and extensible link-time rewriting framework. In *Proc. of the 2005 IEEE International Symposium On Signal Processing And Information Technology* (Athens, Greece, 12 2005), pp. 7–12.

[105] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review 27*, 5 (Dec. 1993), 203–216.

[106] Wang, S., and Malik, S. Synthesizing operating system based device drivers in embedded systems. In *Proc. of the 1st IEEE/ACM/IFIP Intl. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)* (Newport Beach, CA, Oct. 2003), pp. 37–44.

[107] Weber, T. SMT solvers: New oracles for the HOL theorem prover. *International Journal on Software Tools for Technology Transfer (STTT) 13*, 5 (2011), 419–429.

[108] Winwood, S., and Chakravarty, M. M. T. Secure untrusted binaries—provably! In *Workshop on Formal Aspects in Security and Trust* (2005), vol. 3866 of *LNCS*, pp. 171–186.

[109] Xu, Z., Miller, B. P., and Reps, T. Safety checking of machine code. In *Proc. of the ACM SIGPLAN 2000 Conf. on Programming Language Design and Implementation (PLDI)* (2000), pp. 70–82.

[110] Yang, X., Chen, Y., Eide, E., and Regehr, J. Finding and understanding bugs in C compilers. In *Proc. of the ACM SIGPLAN 2011 Conf. on Programming Language Design and Implementation (PLDI)* (2011).

[111] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. Native client: A sandbox for portable, untrusted x86 native code. *Communications of the ACM 53* (Jan. 2010), 91–99.

[112] Yorsh, G., Yahav, E., and Chandra, S. Generating precise and concise procedure summaries. In *Proc. of the 35th Symp. on Principles of Programming Languages (POPL)* (2008).

[113] Yu, D., Hamid, N. A., and Shao, Z. Building certified libraries for PCC: Dynamic storage allocation. *Science of Computer Programming 50*, 1-3 (2004), 101–127.

[114] Yu, D., and Shao, Z. Verification of safety properties for concurrent assembly code. In *Proc. of the Intl. Conf. on Functional Programming (ICFP)* (Sept. 2004).